DIGITAL SYSTEMS LABORATORY

STANFORD ELECTRONICS LABORATORIES
DEPARTMENT OF ELECTRICAL ENGINEERING
STANFORD UNIVERSITY · STANFORD, CA 94305

SU-326-P.39-27

SEL-77-048

THE STRUCTURE OF DIRECTLY EXECUTED LANGUAGES: A NEW THEORY OF INTERPRETIVE SYSTEM DESIGN

by
Lee W. Hoevel
and
Michael J. Flynn

Technical Report No. **130**

March 1977

The work described herein was supported in part by the Department of Energy under contract no. EY-76-S-03-0326-PA 39 and the Army Research Office-Durham under contract no. DAAG-29-76-G-000 1.

SEL - 77- 048

SU-326-P.39-27

THE STRUCTURE OF DIRECTLY EXECUTED LANGUAGES: A NEW THEORY OF INTERPRETIVE SYSTEM DESIGN

by

Lee W. Hoevel

and

Michael J. Flynn

Technical Report No. 130

March 1977

Digital Systems Laboratory

Departments of Electrical Engineering and Computer Science

Stanford University

Stanford, CA 94305

The work described herein was supported in part by the Department of Energy under contract no. N-76-S-03-0326-PA 39 and the Army Research Office-Durham under contract no. DAAG-29-76-G-0001.

SEL-77-048 SU-326-P.39-27

Digital Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305

Technical Report No. 130

March 1977

THE STRUCTURE OF DIRECTLY EXECUTED LANGUAGES: A NEW THEORY OF INTERPRETIVE SYSTEM DESIGN

by

Lee W. Hoevel

and

Michael J. Flynn

ABSTRACT

This paper concerns two important issues in the design of optimal languages for direct execution in an interpretive system: binding the operand identifiers in an executable instruction unit to the arguments of the routine implementing the operator defined by that instruction; and binding operand identifiers to execution variables. These issues are central to the performance of a system, both in space and time.

Historically, some form of "machine language" is used as the directly executable medium for a computing system. These languages traditionally are constrained to a single "n-address" instruction format; this leads to an excessive number of "overhead" instrudtions that do nothing but move values from one storage resource to another being imbedded in the executable instruction stream. We propose to reduce this overhead by increasing the number of instruction formats available at the directly executed language level.

Machine languages are also constricted with respect to the manner in which operands can be "addressed" within an instruction. Usually, some form of indexed base-register scheme is available, along with a direct addressing mechanism for a few, "special" storage cells (i.e., registers, and perhaps the zeroth page of main store). We propose a different identification mechanism--based on the Contour Model of Johnston. Using our scheme, only N bits are needed to encode any identifier in a scope containing less than 2**N distinct identifiers.

Together, these two results lead to directly executed language designs which are optimal in the sense that: (1) k executable instructions are required to

implement a source statement containing k functional operators; (2) the space required to represent the executable form of a source statement contining k distinct functional operators and v distinct variables approaches F*k + N*v -- where there are less than 2**F distinct functional operators in the scope of definition for the source statement, and less than 2**N distinct variables in this scope. (3) the time needed to execute the representation of a source statement containing k functional operators, d distinct variables in its domain, and"? distinct variables in its range approaches d + r + k; where time is measured in memory references.

The work described herein was supported in part by the Department of Energy under contract no. EY-76-03-0326-PA 39 and the Army Research Office-Durham under contract no. DAAG-29-76-G-0001.

1. INTRODUCTION

This report addresses the problem of representing programs for direct machine interpretation. The obvious inadequacies of present machine architectures, in terms of program size and execution time, are well known[1]. Less obvious secondary effects have led to complicated, even Byzantine system structures and implementations[2]. We contend that this is due to the fact that traditional systems are based on the premise that the executable machine architecture must be a fixed and hence universal language. The central thesis of this research is that having to represent programs in a language that is fixed, a priori with respect to system design, forces interpretation to occur at too low a level, places too great a burden on the translation, and limits the potential efficiency of a system.

It is assumed that programs are initially expressed in a higher level source language (HLL), which caters to both the user and the problems that must be solved; but must ultimately be evaluated by a much lower level processor -- the system's host machine. Once the source language and host machine for a system have been selected, the issue becomes one of determining the most suitable intermediate

---

[1]C.f., Flynn [6], Green [11], Lawson [19], Lunde [20], Weber [29], and Wortman [32].

[2]E.g., contemporary compilers, linkage editors, and mechanisms for recognizing and exploiting parallelism -- Sethi [25], and Wichman [30].

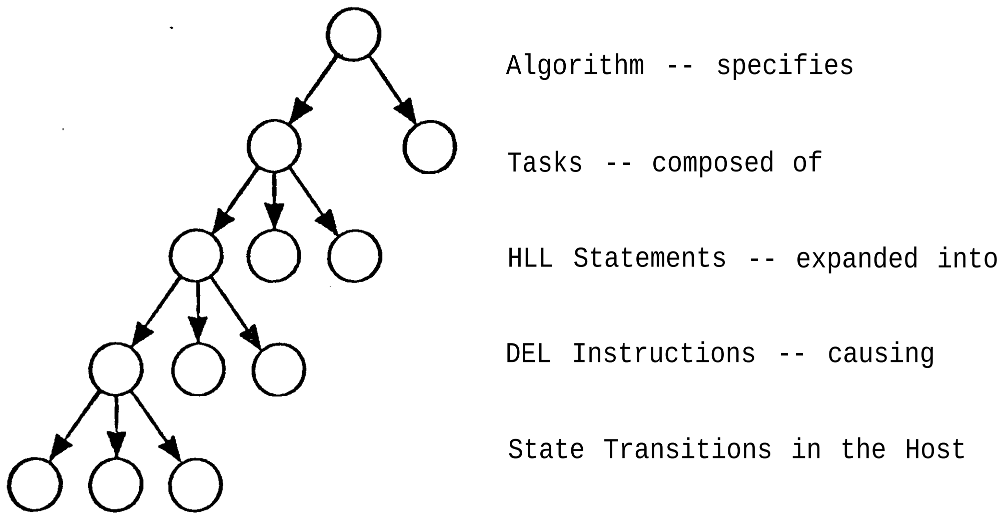
$$\begin{tikzcd}
\text{Abstract Algorithm} \arrow[d] \\
\text{[User]} \arrow[d] \\
\text{Source Program (in HLL)} \arrow[d, phantom, "1"] \\
\text{[Compiler]} \arrow[d] \\
\text{Intermediate Surrogate (in DEL)} \arrow[d] \\
\text{[Interpreter]} \arrow[d] \\
\text{Individual DEL Instruction} \arrow[d] \\
\text{[Execution Semantics]} \arrow[d, phantom, "1"] \\
\text{Host State Transitions} \arrow[d] \\
\text{Evaluation-Process}
\end{tikzcd}$$

Figure 1:

language (or instruction set) for the system -- which we call its <u>directly</u> <u>executed</u> <u>language</u> (DEL). It is important that this intermediate language preserve as much information concerning the user environment and original source program structure as is useful in realizing concise representation and expeditious interpretation (Figure 1).

1.1. A Hierarchial Model

Modelling the evaluation process is complicated by the fact that a computation is actually a hierarchy of interpretations, each level of which may be far more complex than first apparent. Consider the sentence: "An algorithm is defined by a collection of tasks (programs) composed of higher level language statements that are compiled into sequences of lower level instructions, which eventually cause the host machine to undergo a series of state transitions". This describes the five level hierarchy illustrated below:


Algorithm -- specifies
Tasks -- composed of
HLL Statements -- expanded into
DEL Instructions -- causing
State Transitions in the Host

Hierarchial Structure of a Problem

Each level represents the program (algorithm) in a different way; i.e., defines the same process, even though the coding of individual commands is different. The problem of representing programs in an efficient manner begins at the upper most level, and is affected by each of the processes involved in an evaluation. Unfortunately, it is difficult (if not impossible) to recover from faulty program representations at higher levels through sophistocated interpretation techniques at lower levels. This is troublesome, since we would like to minimize both the space needed to represent a program and the time needed to interpret it. Hence, while the significance of uniform formal techniques for defining ideal program representation and interpretation should not be underestimated, this report focuses only on the three lower levels of the hierarchy? it is simply assumed that algorithms are expressed efficiently at higher levels.

1.2. Programs, Instructions, and Computations

At any level of the hierarchy, a program may be defined as a finite set of labelled instructions {I}. Each instruction specifies a pair of rules: an action rule A; and a sequencing rule S. The computation produced by executing a program is defined in terms of a sequence of states where each state denotes a specific assignment of values to program objects. Each action rule defines a function (or operator) f, which takes some number of arguments (dependent on its order) and maps them into (usually) a single result -- arguments and results are, collectively, called operands.

The number of operands in an instruction Ik is fixed and determined by fk. Action rules are often expressed algebraically -- e.g.:

$$y_k = f_k(x_{k,1}, x_{k,2}, \ldots, x_{,k,n})$$

(where n, called the order of f_k, is the number of arguments required by f_k). The number of different functions that can be specified by an instruction set is its vocabulary, or operator set. In general purpose computers, the order of these functions rarely exceeds two, with at most one result being produced.

Each sequencing rule S_k defines the successor to the k^{th} instruction whenever it is executed. In most familiar computer organizations, sequencing is a simple operation -- each instruction having only a single successor. However, specific instructions may require inspection of several arguments before it can be determined which of several possible successors is correct -- e.g., as in the familiar conditional branch instruction.

1.3. Identifiers and Name Spaces

An additional aspect of computation concerns the means by which program objects -- the arguments or result of action rules -- are identified. In general, names are used as surrogates for objects -- which are associated with specific values by the current state of a computation. It is useful to distinguish between the logical name of an object, and the specific encoding of that name appearing in a given instruction -- commonly called an _identifier_.

When an action rule is applied, the encoded names within its instruction must be associated, or bound, to the appropriate program objects. This process is called _referencing_. The set of names for all objects referenced during a computation is called the _process name space_; the set of all identifiers appearing in a program is called the _program name space_. It is important to distinguish between these two concepts: the name space of a process is generally data dependent, and dynamic in nature; the name space of a program is defined by its encoding, and is fully static. Users relate the observable but low level results of executing a program (i.e., the sequence of host machine states produced) to source level semantics through a mental association established between the source level name space and the host name space. The complexity -- and accuracy -- of this mapping determines the ultimate transparency of a system.

2. TOWARDS IDEAL PROGRAM REPRESENTATIONS [8]

By what criteria should program representations be judged? Clearly, an efficiency measure should lie in some sort of space-time product involving both the space needed to represent an executable program and the time needed to interpret it; although other factors -- such as the space and time needed to create executable representations, or the space needed to hold the interpreter -- may also be important. This report considers only the space and time needed to represent and execute a program.

2.1. Canonic Interpretive Forms

Characterizing "ideal" program representations can be either trivial or extremely complicated, depending on one's point of view. Neither extreme offers significant insight into the problems at hand, however. It is therefore imperative to develop constructive space-time measures that can be used to explore practical alternatives. Although these measures need not be achievable, they should be satisfied only by clearly superior representations, easy to define, easy to use, and in clear agreement with both a programmer's intuition and pragmatic observations. We propose the following canonic interpretive form, or CIF, as a measure of statement representation in a high level programming language.

1:1 Property

Instructions -- one CIF instruction is permitted for each non-assignment type operation in a HLL statement.

Name Space -- one CIF name is permitted for each unique[3] HLL name in a HLL statement.

Log_2 Property

Instructions -- each CIF instruction consists of:

A single operation identifier of size $[\log_2(F)]$[4]; and one or more operand identifiers, each of which is of size $[\log_2(V)]$[5].

Referencing Property

Instructions -- each HLL procedural (program control) statement causes one canonic reference.

Name Space -- one reference is allowed for each unique variable or constant in the HLL statement.

Space is measured by the number of bits needed to represent the static definition of a program; time by the number of instructions and name space references needed to interpret the program. Source programs to which these measures are applied should themselves be

---

[3] I.e., distinct name in the HLL statement; "A = A+1" contains two unique names -- the variable "A" and the constant "1".

[4] F is the number of distinct HLL operators in the scope of definition for the given HLL statement.

[5] V is the number of distinct HLL program objects -- variables, labels, constants, etc. -- in the relevant scope of definition

efficient expressions of an optimal abstract algorithm -- so as to eliminate the possible effects of algorithm optimization during translation -- such as changing "X = X/X" to "X = 1."

Generating canonic program representations should be straight forward because of the 1:1 property. Traditional three address architectures[6] also satisfy the first part of this criteria, but do not have the unique naming property.

For example, the statement "X = X + X" contains only one unique variable, and hence can be represented by a single CIF instruction consisting of only one operation identifier and one operand identifier. The three address representation of this statement also requires only a single instruction, but it would consist of four identifiers rather than the two required by the CIF.

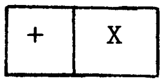
There may be some confusion as to what is meant by an "operation". Functional operators (+, -, *, /, SQRT, etc.) are clear enough; however, allowance must also be made for selection operators that manipulate structured data. For instance, we view the array specification "A(I,J)" as a source level expression involving one operator (two dimensional qualification) and at least three operands

----

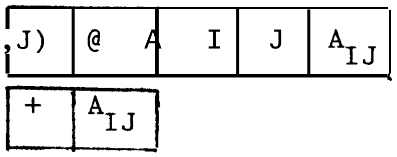
[6]I.e., instruction sets of the form OP X Y Z -- where OP is an identifier for a (binary) operation; X the left argument; Y the right argument; and Z the result.

(the array A, and its subscripts I and J). Therefore, unlike the previous case, the canonic equivalent of "A(I,J) = A(I,J) + A(I,J)" requires two instructions -- the first to select the proper array element, and the second to compute the sum. Thus:

Example 1: X=X+X



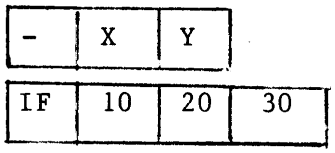
Example 2: A(I,J) = A(I,J) + A(I,J)



The operator "@" computes the address of the doubly indexed element "A(I,J)", and dynamically completes the definition of the local identifier "A_{IJ}". This identifier is then used in the same manner as the identifier "X" is used in the first example.

We count each source level procedural operator, such as IF or DO, as a single operator. The predicate expression of an IF must, of course, be evaluated independently if it is not a simple variable reference. Distinct labels are treated as distinct operands , so that:

Example 3: IF (X-Y) 10,20,30



Two accesses to the process name space (references) are required to execute the first example: one to fetch the value of X as an argument, and one to update its value as a result of executing the statement. In example two, four references are required: one each to fetch the values of I and J for the subscripting operation; one to fetch the value of A_{IJ} as an argument; and one to update the value of this array element after execution. Note that no references are required to access the array A, even though it appears as an operand of the @ function -- in general, no single identifier in a CIF instruction can cause more than one reference unless it is bound to both an argument and a result, and then it will initiate only two references. No references are needed for either example just to maintain the instruction stream, since the order of execution is entirely linear[7]. The 1:1 property measures both space and time, while the $\log_2$ property measures space alone, and the referencing property measures time alone. These measures may be applied either statically or dynamically -- although static reference counts are strictly comparative, and hence of limited value.

The 1:1 property defines, in part, a notion of _transformational completeness_ -- a term which we use to describe any intermediate language satisfying the first canonic measure. Translation of source programs into a transformationally complete language should require neither the introduction of synthetic variables, nor the insertion of

---

[7]The assumption here is that such reference activity can be fully overlapped since it is so predictable.

non-functional memory oriented instructions[8]. However, since the canonic measures described above make no allowance for distinguishing between different associations of identifiers to arguments and results, it is unlikely that any practical language will be able to fully satisfy the CIF space requirements.

2.2. Comparison of CIF to Traditional Machine Architectures

Consider the following three line excerpt from a FORTRAN subroutine:

```
1     I = I + 1
2     J = (J-1)*I
3     K = (J-1)*(K-I)
```

Assume that I, J, and K are fullword (32 bit) integers whose initial values are stored in memory prior to entering the excerpt, and whose final values must be stored in memory for later use before leaving the excerpt. The canonic measures for this example are:

----

[8]E.g., to hold the results of intermediate computations, or move data about within the storage hierarchy merely to make it accessable to functional operators.

CANNONIC MEASURE OF THE FORTRAN FRAGMENT

Instructions

Statement 1 --	1 instruction	(1 operator)
Statement 2 --	2 instructions	(2 operators)
Statement 3 --	3 instructions	(3 operators)

Total	6 instructions	(6 operators)

Instruction Size

Identifier Size

Operation identifier size = $\lceil \log_2 4 \rceil$ = 2 bits
(operations are: +, -, *, =)

Operand identifier size = $\lceil \log_2 4 \rceil$ = 2 bits
(operands are: 1, I, J, K)

Number of Identifiers

Statement 1 --	3 identifiers	(2 operand,	1 operator)
Statement 2 --	5 identifiers	(3 operand,	2 operator)
Statement 3 --	7 identifiers	(4 operand,	3 operator)

Total	15 identifiers	(9 operand,	6 operator)

Program Size

6 operator identifiers x 2 bits =	12 bits
9 operand identifiers x 2 bits =	18 bits

Total	30 bits

References

Instruction Stream --	1 reference	(nominal)
Operand Loads --	9 references	
Operand Stores --	3 references	

Total	13 references	

The following listing was produced on an IBM System 370 using an optimizing compiler[9]:

```
1   L     10,112(0,13)
    L     11,80(0,13)
    LR    3,11
    A     3,0(0,10)
    ST    3,0(10)

2   L     7,4(0,10)
    SR    7,11
    MR    6,3
    ST    7,4(0,10)

3   LR    497
    SR    493
    LCR   393
    A     3,8(0,10)
    MR    2,4
    ST    3,8(0,10)
```

A total of 368 bits are required to contain this program body (we have excluded some 2000 bits of prologue/epilogue code required by the 370 Operating System and FORTRAN linkage conventions) -- over 12 times the space indicated by the canonic measure. Computing reference activity in the same way as before, we find 48 accesses to the process name space are required to evaluate the 370 representation of the FORTRAN excerpt. If allowance is made for the fact that register accesses consume almost no time in comparison to accesses to the execution store, this count drops to 20 references -- allowing one access for

[9]FORTRAN IV level H, OPT = 2, run in a 500K partition on a Model 168, June 1977.

each 32 bit word in the instruction stream.

The increase in program size, number of instructions, and number of memory references is a direct result of the-partitioned name space, indirect operand identification, and restricted instruction formats of the 370 architecture. In order to facilitate the discussion at this point, it is useful to define [6] three general classes of instructions:

M-instructions, which simply move data items within the storage hierarchy (e.g., the familiar LOAD and STORE operators);

P-instructions, which modify the default sequencing between instructions during execution (e.g., JUMP, BRANCH and LINK operators); and

F-instructions, which actually perform functional computations by assigning new values to result operands after transforming the current values of argument operands (e.g., all arithmetic, logical, and shifting operators).

Instructions that merely rearrange data accross partitions of a memory name space, or that alter the normal order of instruction sequencing, are "overhead" in the sense that they do not directly contribute to a computation. The ratio of these overhead instructions (i.e., M- and P- type instructions in our terminology) to functional instructions (F-instructions) is indicative of the use of an architecture. Overhead instructions must be inserted into the desired sequence of F-instructions to match the computational requirements of the original program to the capabilities of the machine architecture. Statically, M-instructions are by far the most common overhead instructions -- indeed, they are the most common type of instruction in almost all existing machines. Dynamically, however, P-instructions become equally significant.

The table below illustrates the use of ratios for the foregoing example.

COMPARISON FOR THE EXAMPLE

	370 FORTRAN-IV (level H extended) optimized	370 FORTRAN-IV (level H extended) non optimized	CIF
No. of Instructions	15	19	6
M-type Instructions	9	13	0
F-type Instructions	6	6	6
M-ratio	1.5	2.7	0
Program Size	368 bits	604 bits	30 bits
Memory References	20	36	13

3. DEL SYNTHESIS

This section addresses the problem of designing high performance DEL's. We focus on three particular areas:

Sequencing, which has two aspects --

a. Sequencing between actions (program control).
b. Sequencing within an action (context).

Action Rules, which also have two aspects --

a. The format or transformation used by the rule.
b. The operation invoked.

Name Space, which addresses two issues --

a. Name structure -- the syntax and semantics of identifiers.
b. Name environment -- referencing of variables and operators.

Each of these areas will be reviewed following a statement of term definition and assumptions.

3.1. Terms and Assumptions

In order to synthesize simple "quasi-ideal" DELs, let us make some obvious assignments and assumptions.

* The DEL program representation lies in the main storage of the host machine

* The interpreter for the DEL lies in a somewhat faster, smaller interpretive storage. The interpreter includes the actual interpretive subroutines as well as certain parameters associated with interpretation.

* Only a small number of registers exist in the host machine that can be used to contain local and environmental information associated with the interpretation of the current DEL instruction. Further, it is assumed that communications between interpretive strorage and this register set can be overlapped (Figure 2(a)).

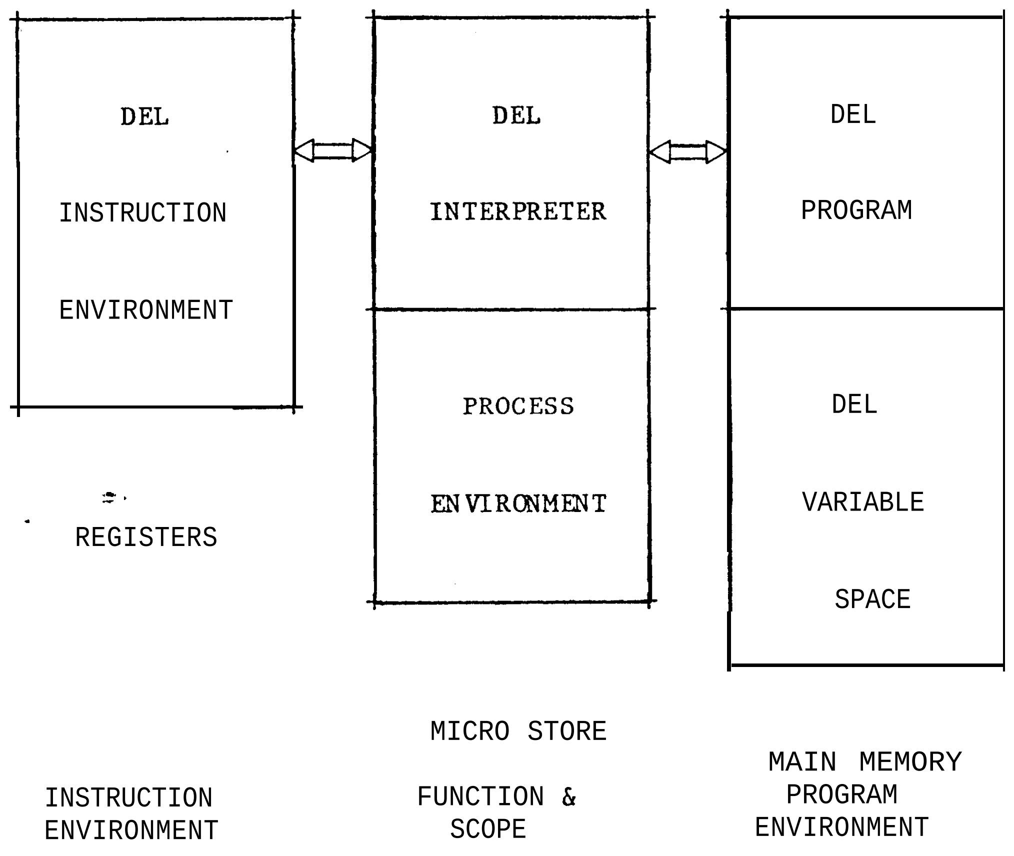


Figure 2(a): DEL/Host Storage Assignment

An instruction is a binary string partitioned into identifiers under action of the interpretive program. An identifier is an element of the vector bit string specifying one of the following:

i. format and (implicitly) the number of operands

ii. the operands

iii. operations to be performed (of at most binary order) on the identified operands

iv. sequencing information, if required.

A format is a rule defining:

1. the instruction partition (i.e. number and meaning of identifiers).

ii. the order of the operation (i.e., whether the operation is in nullary, unary or binary).

iii. precedence among operands (i.e., binding of operand identifiers to functional operands).

In this report, it is assumed that DEL instructions are use ordered -- i.e., that the internal sequence of identifiers within an instruction is the same as the sequence in which these identifiers will be required during interpretation. The 370 architecture is not use ordered, since the format/operation code appears before operand identifier information. This forces the interpreter to "save" the operation code during computation of effective addresses -- wasting, at least temporarily, a scarce host register.

The size of an identifier is the width of the field it occupies within an instruction. It is determined by the number of elements required in a locality; the structure of a typical DEL instruction is illustrated in Figure 2(b).

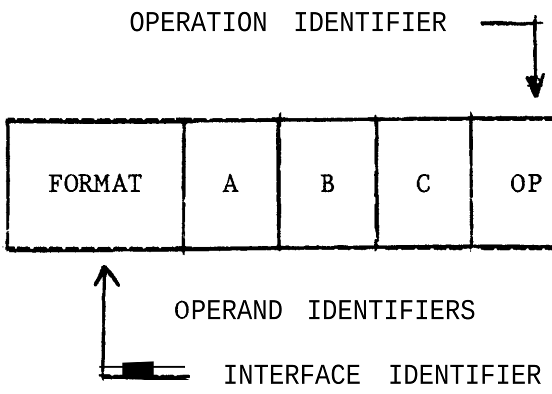


Figure 2(b): Layout of a Typical DEL Instruction

3.2. Sequencing Rule

Usually, a program consists of a sequence of action rules. The sequecing rule provides the ordering relation among the action rules -- i.e., it defines the sequence of the action. While it is possible to conceive of DEL's with unordered action rules (no sequence rule), this form is of little value.

3.2.1. Sequencing Between Actions

In practice only a few sequencing rules have been used with any degree of success. We consider the following three rules:

<u>Linear</u>: individual instructions are stored in a one dimensional array within the main store. Execution order is the same as the array ordering unless modified by a branch instruction.

<u>Binary Tree</u>: instructions are mapped into the nodes of a tree structure maintained in main store. Leaf nodes normally correspond to data references; ancestor nodes to semantic functions. A standard traversal algorithm defines the default order of execution, which can be modified by visiting a branch node.

<u>Linked List</u>: instructions are stored at the links in a chain structure maintained in main store. The default execution order is again specified by a traversal algorithm, and can be modified by the semantics associated with the most recently visited link.

These three forms are abstracted from well known programming structures. Most traditional machine language DELs are based on a linear form. Tree form are widely used as intermediate data structures by compilers. Linked lists are the fundamental program and data structures for LISP and PPL (McCarthy [21], and Standish [26]). Tree and list data structures are widely used in the algorithms employed in artificial intelligence and information retrieval applications. Figure 3 illustrates program representations in the linear, tree, and list forms.

The particular DEL organization used in these examples is arbitrary, for purposes of illustration only, and is not necessarily optimal. Similarly, neither the operators nor data structures are completely specified; they should be assumed to have the same general interpretation for all three DEL forms. These fragments are constructed so that the order of execution will be identical (i.e., the sequence of functional operations and storage accesses will be the same).

Figure 3:   Three Representations of " I = J * (K + L) ; "

```
(a) -- Linear            push @I
                         push J
                         push K
                         push L
                         + (add)
                         * (multiply)
                         = (assign)
```

```
(b) -- Tree                    { = }
                             /       \
                            |         |
                            v         v
                          { I }     { * }
                                   /      \
                                  |        |
                                  v        v
                                { J }    { + }
                                        /     \
                                       |       |
                                       v       v
                                     { K }   { L }
```

```
(c) -- List

 +-------  +-------  +-------+
 | = : --->| . : --->| .   0 |
 +-------  +-|-----  +-|-----+
            /
        +-v-----+  +-v-----  +-------  +-------  +-------  +-------+
        | I : 0 |  | * [illegible]                    .            |
        +-------+  [illegible]     +-|----- +-------- +-|----- +-|------+
                                     |                 /        |
                               +-v-----+         +-v-----+  +-v-----+
                               | J : 0 |         | K : 0 |  | L : 0 |
                               +-------+         +---mm--+  +-------+
```

3.2.1.1. Linear Forms:

The sequencing rule for a DEL governs the way in which control is passed from one instruction to another. If a linear form is used, for example, the normal sequence of execution is implied by the placement of DEL instructions within the main store. A program counter is usually maintained within the interpreter, as part of the DEL program status vector, which points to the word containing the next DEL instruction to be executed. When the contents of the current instruction word are interpreted, the word pointed to by the program counter is fetched, the counter incremented appropriately, and execution continues. Interpreting a branch instruction causes the DEL program counter to be loaded with a new address that points to the next instruction to be executed. The set of branching instructions in a DEL is not confined to the simple GOTO, but may also include more complex program control operators such as CALL, RETURN, DO, and IF-THEN-ELSE.

Since the default sequencing rule for a linear DEL is to simply process the instruction stored "immediately after" the one just executed, there is a good match between this form and cyclically addressable main stores. This can be exploited by carefully packing DEL instructions so that the essential fetch and sequence steps within the basic cycle of interpretation can be implemented efficiently. This can almost always be achieved with minimal execution time overhead using only elementary shift and increment capabilities.

The natural ordering of addressable storage cells can be used to induce a default order of interpretation, thus eliminating the need for explicit sequencing of pointers in linear segments of DEL code. As individual instructions are more highly compressed, fewer main store accesses are required to maintain a given DEL instruction stream. For example, suppose that each instruction in a linear DEL contains the address of its successor as an explicit subfield. An interpreter would sequence through instructions by fetching the successor address from the instruction just executed, and then obtaining the next instruction to be executed from that address in main store. No internal program counter need be maintained unless relative branching is required.

This DEL could be made more efficient by eliminating explicit successor addresses within instructions that do not cause a branch out of the normal linear order. An interpreter for this new DEL must maintain an internal program counter that is updated by the length of the current instruction during each cycle of interpretation. However, program representations will be smaller -- and should be faster -- than those of the previous DEL, assuming that main store is sufficiently slower than micro store.

3.2.1.2. Tree Forms:

Tree structures are used by many compilers as an intermediate form from which the final, executable code is generated. Intuitively, ancestor nodes refer to operators (non-terminals in the source language syntax), while leaf nodes refer to variables (syntactic terminals). The operation code associated with a node is combined with two or tree pointers to form a unit of fixed, uniform size. These units constitute the phywical realization of a tree structure within the main store of the host machine. The units for a binary tree DEL need contain only two pointers in a minimal realization: (1) the address of the unit for the left descendent of a node; and (2) the address of the unit for its right descendent.

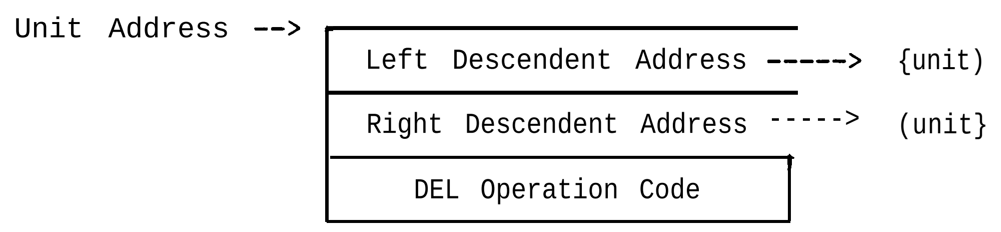

Unit Address -->
Left Descendent Address -----> {unit)
Right Descendent Address -----> (unit}
DEL Operation Code

Figure 4: Typical Binary Tree Unit

The left and right descendents of an ancestor node which is associated with a binary operator correspond to its left and right operands, respectively. Usually, the operators in a DEL are binary if a tree structure form is selected -- unary operators are treated as degenerate binary operators, with null right descendent pointers. Some auxiliary pointers (usually to the ancestor of a node) may be included to facilitate tree traversal, however.

Perhaps the most widely used traversal strategy is 'depth first, left to right postorder" -- meaning that a node is executed only after both its left and right descendents have been evaluated. Under this rule, successive left descendents are visited until a "left value" is computed, then the right descendent is visited (Knuth [18]). Only after both the left and right values of a node are known will the node itself be visited. Finding the unit for a successor node is a simple matte'?; at least when traversing downward. Only a primitive load operation is required at the micro level to extract the address of the proper descendent unit, so DELs based on a tree form are easily interpreted by a wide range of microprogrammable hosts.

There is a significant problem with the obvious implementation of this algorithm, however: the interpreter must maintain a stack of pointers to nodes that have been visited, but not yet executed. Entries in this stack are the addresses of units associated with non-terminal nodes that must be reexamined after computing the values of lower level nodes. Maintaining this stack enlarges the interpreter state and complexity. The need for this stack can be eliminated, at the expense of DEL program space, by including a "back pointer" in each unit that is the address of immediate ancestor.

One potential advantage of tree DELs is that they are easy to modify incrementally -- i.e., a surrogate can be made to reflect small changes at the source level without a full recompilation. The new subtree produced by recompiling only the affected portion of the source program. This usually requires that program control transfer points and DEL variables be identified by node rather than address, and may also necessitate a run time "garbage collector" to reclaim the holes left by excised DEL code.

Another potential advantage is that the interpretation of a subtree can be bypassed during an execution if either: (1) the value computed the last time its root node was visited is retained in the root's unit; and (2) none of the values associated with the leaves of the subtree has been modified since the root was last visited. In order to obtain this advantage, though, a complex tagging scheme to mark the validity of the values stored in ancestor units may be needed. Unfortunately, the overhead of such a tagging scheme (incurred each time a node is visited), together with the time required to store the last computed values, may be greater than the time saved by escaping the evaluation of some subtrees. It is not easy to evaluate the tradeoffs involved, though, since adequate statistics are not easily obtained. This strategy at least offers the possibility that tree DELs can be developed which are effectively more compact and more efficient to interpret than linear DELs.

3.2.1.3. List Forms:

The simplest examples of linked lists look much like unary or binary trees; in fact, most of the above tree related comments are equally applicable to linked list DELs. However, the links within a list (its nodes) may be their own ancestors -- i.e., cycles are allowed. Again, instructions are associated with the links in a list representation. They contain a pointer to a successor link, and either an atomic value or a pointer to a value link. A unique pointer, NIL ("0" in Figure 3(c)), is used as the successor pointer in such terminal links.

This classic definition is easily extended to cover lists in which links may reference multiple successor or value cells, thus reducing the number of links needed to represent complicated control and data structures. Traversal usually proceeds by value first, then successor -- analogous to depth first, left to right postorder tree traversal.

Because of their generality, linked lists are not easily address encoded. While the relative spatial cost of link pointers depends on the average size of a DEL instruction; a linked list DEL almost always requires more space than an equivalent linear form DEL, barring extensive factoring of common sublists. However, the marginal cost of incorporating additional address references is low for a linked list DEL representation, and hence it is comparatively easy to implement complex operators that do not easily fit in the binary operator order.

For example, the target of any branch can be directly encoded as one of the successor pointers in its link unit, and need not be treated as an indirect operand. This is not always possible in a tree DEL, since cycles are not allowed. The flexibility of a linked list form can also be exploited by linking units in precisely the order in which they should be interpreted during execution. By converting the linked list in Figure 3(c) into a polish suffix form, for example, backtracking during interpretation could be eliminated. This reduces both the internal state size and complexity of the interpreter, but is not compatible with the factoring technique described above.

In most cases, the pointers required by tree and list structures makes them less desirable than the linear array as a potential DEL form: both because of the space these pointers occupy, and because of the extra main store access needed to determine the location of successor instructions. It is usually far faster to increment a DEL program counter (normally maintained in a host register) than to fetch an address from main store. Unless the flexibility of tree and list forms can be exploited in an innovative manner, the spatial and temporal overhead associated with this single negative aspect may be of overriding importance in selecting the form for a DEL.

3.2.2. Sequencing Within an Action: Context

Defining a sequence rule within an action is primarily a problem of exploiting execution context during an action rule interpretation. Context information may be used to significantly improve action rule representation at the expense of some additional complexity in the interpretation process. We consider five distinct types of context.

3.2.2.1. No Dependencies

The simplest program representations involve no dependencies, and an example of such DELs is "threaded code" -- in which each field occupies a full word of storage, and is itself a direct pointer to either a cell in the DEL data store (operand references) or to a semantic routine in micro store (operator references). This straight forward encoding may in fact be optimal if the host has little or no field extraction capability, since each syllable starts on a word boundary and need not be processed before use during interpretation.

Threaded code programs are similar to highly subroutinized host programs in which there is one subroutine for each semantic routine within the threaded code interpreter. However, CALL and RETURN operators are omitted in the threaded code, which reduces its program store requirements; the interpreter performs the function of the deleted operators. Operands are usually passed as in-line vectors of addresses, and operations indicated by explicit micro store addresses, though, just as arguments are imbedded in the calling sequences of a host machine.

The time needed to fetch a threaded code instruction, in main memory accesses, is k+1; where k is the average number of operands per instruction. If we let b denote the number of bits per word of storage, then the space required to represent a threaded code instruction is b * (k+1).

3.2.2.2. Memory Dependencies

Given a word oriented host, we view instructions as fixed length "records" containing a fixed number of subfields at known boundaries. In this case, use ordering is of minimal importance, since the syllable positions are always known. Selecting an optimal instruction layout is basically an alignment problem; instructions should be stored on bit addresses that minimize the number of main store accesses required to extract critical fields. This problem is examined from the perspective of the computer architect in Flynn and Henderson [7].

Their analysis can be applied directly to the DEL synthesis problem, although there are fewer free variables in this case since the host machine is an assumed given. The relevant result is an analytic expression for the average number of accesses required to retrieve a group of F characters with character address I into a record of length L.

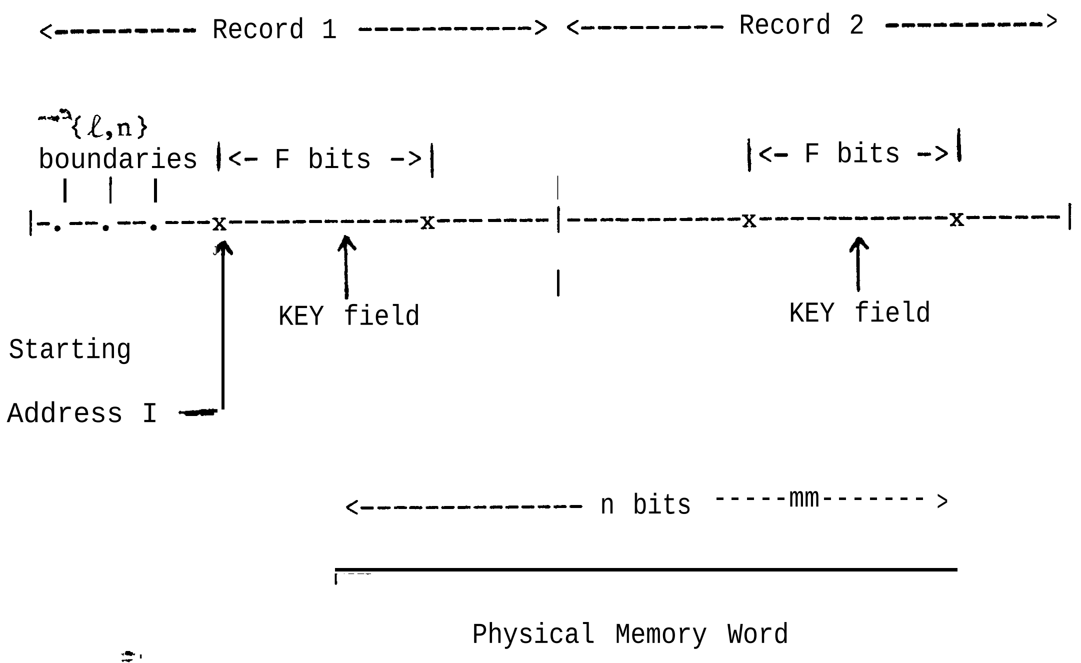


Figure 5: Accessing KEY Fields in DEL Instructions

The group of F characters can be thought of either as an entire DEL instruction -- in which case the notion of a record also corresponds to an instruction -- or as a critical syllable (e.g., the KEY code) within an instruction. In the latter case, the instruction is itself the L character record. If each main store access retrieves n characters of data, the number of accesses needed to fetch the critical portion of an instruction is

$$\text{Accesses} = \left\lceil \frac{F}{n} \right\rceil + \frac{\left\lceil \frac{i+f}{\{\ell,n\}} \right\rceil - 1}{n/\{\ell,n\}}$$

where: f = F Mod n (least positive residue; i.e., x Mod x = x), ℓ = L Mod n (least positive residue); i = I Mod $\{\ell,n\}$(least residue, including 0), and $\{\ell,n\}$ = greatest common divisor of ℓ and n.

Although formidable in appearance, this equation is not difficult to interpret. Clearly, the number of accesses required to fetch a DEL instruction of length F from a unit of length L will be either $\lceil F/n \rceil$ or $\lceil F/n \rceil + 1$, depending on the number of word boundaries crossed. This is determined by the starting address of the instruction. The second term is an analytical representation of the average effect of this placement, assuming that fields occupy integral multiples of the basic storage quantum (e.g., eight bit bytes for a 360/370 environment). While this is a reasonable assumption for a machine designer, character size is often a free variable to the DEL designer (Hoevel and Wallach [13]).

If the host is strongly biased toward a particular character size, then it is probably best to use this as the basic storage quantum for DEL encodings. If the host is unbiased, however, the size of a character should be selected to minimize F/n. The Flynn-Henderson equation shows that it is best to start instructions on character addresses that are integer multiples of $\{\ell,n\}$. In this case, the time needed to fetch a typical DEL instruction, in main storage accesses, is:

$$\text{Access Time} = \lceil F/n \rceil + \frac{f - \{\ell,n\}}{n}$$

while the space needed to represent it is:

$$\text{Program Size} = \ell * b/n = w * (k+1) \text{ bits}$$

As above, k is the number of syllables that must be fetched and decoded to execute the entire instruction, and b is the number of bits per word; w is the average number of bits per syllable.

In most cases F is less than n, and so the average fetch time is minimal-when F is minimized -- i.e., when pointers and/or instructions occupy as few characters as possible. Decoding algorithms for this type of DEL are usually straight forward. Since instructions are word aligned, the exact bit offset of each subfield is known, and decoding is at worst a simple combination of mask and shift operations.

In some cases, special features of the host can be exploited -- such as the transform board capability of the CDC 5600 series, which allows the contents of a micro register to be "exploded" (i.e., distributed accross several other micro registers in a single micro instruction). This board must be physically rewired for each such explosion desired, however, and cannot be changed dynamically during an emulation (Control Data Corporation [4]).

3.2.2.3. Inter Instruction Dependencies

Both the sequence in which instructions are encountered and their placement can affect their interpretation for certain DELs. The primary reason for selecting a form with inter instruction dependencies is to minimize the size of a typical DEL program, and thus indirectly reduce the average fetch overhead. Since a relatively large space penalty is usually incurred when a tree or list sequencing rule is used, these forms are most often applied to linearly sequential DELs.

To exploit the similarity between integer addressable stores and locally sequential program structure, a design permitting multiple DEL instructions to be placed in a single word of storage must be devised. Minimizing the size of individual DEL instructions is quite important here, although if an execution time advantage is to be realized the encoding must be simple to recognize and decode.

Usually, the DEL program state vector is augmented so that the interpreter can remember unused, but previously fetched portion of the DEL instruction stream. Specifically, a residual control cell called the <u>current instruction word</u> (IW) is needed. This word contains those bits in the DEL instruction stream that were brought into host storage registers during the last instruction stream access to main store, but which have not been decoded.

This type of dependency is most effective for hosts with wide storage resources and a large ratio between main and micro store bandwidths. To a first approximation, if an average of m instructions can be packed into a single word, the time needed to fetch a given instruction stream may be reduced by a factor of m compared to a fully independent technique.

Interpreters for instruction stream dependent DELs must maintain at least two elements of residual control: a DEL program counter (PC); and current instruction word (IW). If full prefetch is implemented, and additional residual control cell is needed -- a successor instruction word (SW). The interpreter attempts to maintain the next word of instruction stream bits in SW (i.e., keep SW equal to the contents of the successor to the word last loaded into the IW). When all of the bits in the IW have been decoded, its contents are replaced by the contents of SW, the PC is updated, and most of the time needed to transfer instruction words from main store into the internal resources of the host to be overlapped, but this implies that the PC, IW, and SW must be maintained in the fastest storage resowrce (i.e., host registers). Use ordering of syllables is important in a strongly context dependent DEL, since such a large fraction of the micro level storage resources must be dedicated to maintaining the DEL instruction stream.

For example, decoding an operator specification prior to the specifications of its operands (as in the natural sequence of interpretation for the 360/370 architecture) forces the interpreter to store the operator code across the operand fetch portion of the interpretation cycle. This both lengthens execution time and increases interpreter size. Also, instructions need not be word aligned. This means that it may be more difficult to decode the syllables, since it can no longer be assumed that they are aligned on specific address boundaries.

If the host has a register pair shift capability, a K bit internal field extraction may be accomplished by register pair shifting K bits from the retained instruction stream word into a previously cleared index register (IX). If the host has only a single word shift capability, then both a mask and shift are required. Both of these techniques are illustrated below.

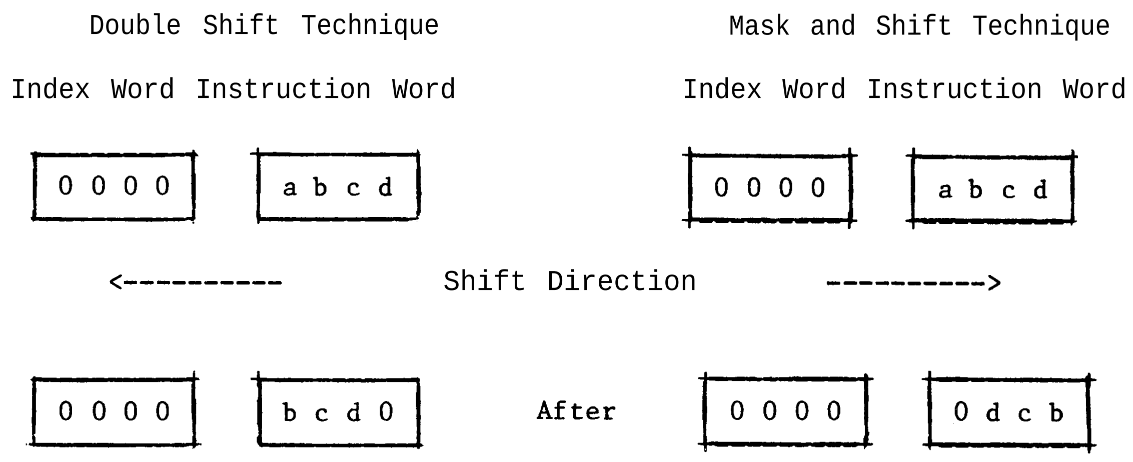


Figure 6: Before and After Snapshots of a Syllable Extraction

In this diagram, lower case letters denote specific codes for individual syllable codes, and the "mask" is zero except at bit positions occupied by the syllable code being extracted (i.e., "a"). Although shift direction is critical in the register pair shift technique, it is easy to develop a mask and shift strategy for hosts posessing only a single left circular shift.

3.2.2.4. Memory Mapping and Word Boundary Dependencies

For the moment, assume that a DEL instruction consists of a sequence of as yet undifferentiated syllables. These syllables may be of a single, uniform width (often the case for polish DELs), any of a fixed number of different widths, or even of dynamically varying widths. Consider the following three strategies for coping with these possibilities:

i. Dynamically concatenate successive words in the DEL program store, in effect creating a "bit stream" memory.

ii. Code the fact that the next n syllables lie within the current instruction word as part of the semantic interpretation of the first (or last) syllable in the instruction.

iii. Reserve one syllable code (usually all zeroes) to signify "end of instruction word" -- i.e., that the current instruction word is exhausted (i.e., has been interpreted), and a new instruction word fetch is required.

The first technique is used in the Burroughs S-language implementation for the B1700, a defined field host capable of accessing arbitrary sized fields at bit addresses. By packing DEL instructions at the bit level means that "every bit is fully utilized", and "appears to account for half of all the program compaction which has been realized on the B1700" (Wilner [31]).

There can be a high interpretation time penalty associated with frequency encodings, however, since several sequential levels of decoding may be required to correlate a syllable code with the proper semantics. Wilner outlines an "SDL" encoding that is claimed to obtain most of the compaction resulting from Huffman's code [14], while still permitting reasonable decode times. The resulting polish form instructions are about thirteen bits in length (averaged over both operator and data instructions), and require a maximum of three stages of decode. Wilner estimates that a pure Huffman code would be fourteen per cent slower to decode, but would only reduce the size of a typical surrogate by one per cent.

These time estimates may be unique to the B1700 and the specific interpretation algorithm used to process the S-languages. Although Wilner claims only a 2.6 per cent slow down from a straight n-way binary code to a 4-6-10 staged encoding, the manner in which this is computed is not clear. It may be that little or no retention is used by S-language interpreters, or that instruction fetch time is included in the computation of decode time -- which would certainly tend to equalize differences between various techniques. Decoding SDL codes on an EMMY [24] based system would require more than double the time needed by a simple n-way binary code. This is equivalent to more than 40 per cent of a typical instruction execution; if a pure Huffman code were used, this factor could register pair again. At least some direct hardware assistance appears to be necessary for this technique to achieve high performance.

The second strategy is nothing more than the familiar fixed field organization used by most second and third generation "machine languages". Once the first few bits of such a DEL instruction have been decoded, the exact length and placement of all the subfields within that instruction can be determined. In this case, the Flynn-Henderson equation can be used to adjust the overall length of the various instruction types so as to minimize the time needed to fetch a given instruction stream -- i.e., minimize the time needed to access the critical fields that define the transformations to be performed.

An interesting variation of this scheme is used for CRIL [15], in which the semantics associated with the operation defined by an instruction specify whether or not the next instruction to be executed lies within the same word of storage as the current instruction. In general, the successors to arithmetic operations lie in the same word, while successors to conditional branches lie in the storage word at the next higher address (assuming the branch is not taken -- see ICL [15]). The 360/370 "fixed format" inner form results in an average instruction size of about 24 bits; the ICL approach reduces this to about 20 bits, while maintaining the same relative instruction set capability.

The last technique was developed independently during the synthesis of DELtran (Hoevel [12]). It approximates the bit stream packing capability of the B1700, but requires only two registers, the instruction index IX and instruction word IW, and is easily implemented on hosts with flexible memory arrangements. Each DEL instruction is treated as a string of syllables that is fetched and decoded as follows:

1. A syllable is extracted from the IW using either of the two methods described above.
2. If the IW is now zero, transfer of the next word in the instruction stream into the IW is initiated.
3. The appropriate routine is invoked, depending on the contents of the IX, and execution continues with step one.

Using this technique, the all zeros code must be reserved to indicate that the current instruction word has been exhausted, which is not true for the SDL bit packing. However, the zero code strategy can be implemented without increasing the size of the interpreter state (either the IW or IX registers may be tested for equality with zero after extracting a syllable), and a minimal number of host instructions are involved. In constrast, a seperate bit position counter is required to properly concatenate successive SDL syllables in hosts like the EMMY and CDC 5600, and extra host instructions may be needed if the host is not sufficiently parallel.

The generation algorithm for this is to simply place successive syllable codes into a word until the next code does not fit within that word. The current word is then filled with zeros, and the process is repeated for the next word in the DEL program store.

The following is a simple technique, hinging on the definition of "fit", that can save some execution phase time and space. Suppose that there are M bits in the next syllable code to be packed into a word that has only N bits remaining, where M is greater than N. The first N bits of this syllable can be packed into the current word if its M-N trailing bits are zero -- they will be supplied automatically by the algorithm outlined above. This results in individual syllables being logically, if not physically, contained within individual program store words, but permits entire instructions to cross word boundaries.

By assigning these codes such that frequently occuring codes have a greater number of trailing zeros, the beneficial effects of this technique should be significantly improved. The information capacity of any given syllable is decreased by the mandatory "all zeros" code only if there are exactly 2^w other alternatives that must be distinguished by its content, where w is the bit width of the syllable.

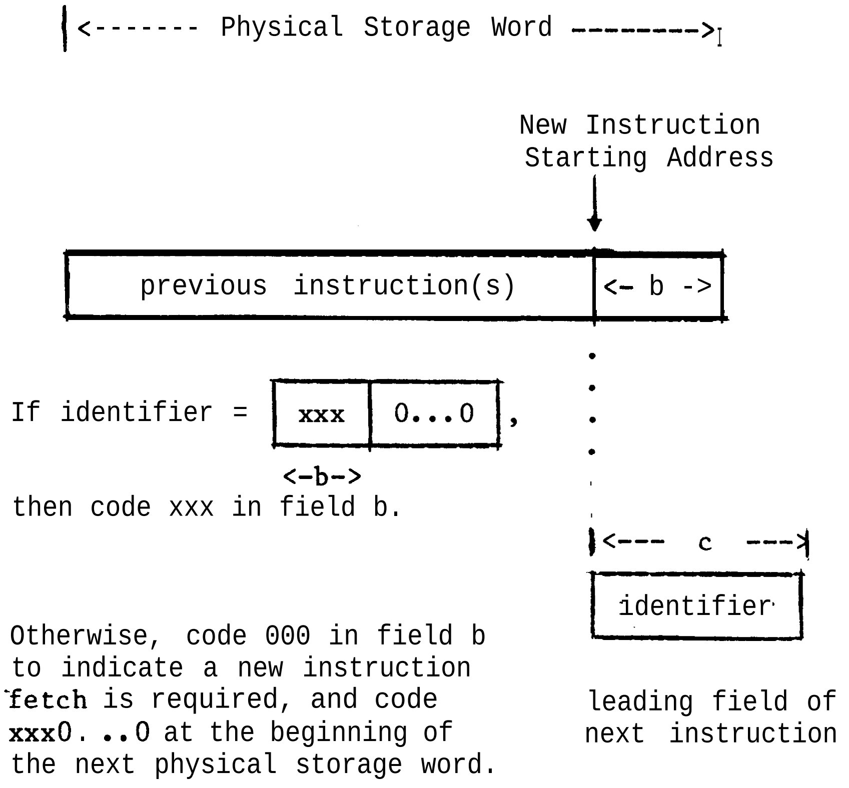

|<------- Physical Storage Word -------->|
New Instruction Starting Address
previous instruction(s)
<- b ->
If identifier =
xxx
0...0
<-b->
then code xxx in field b.
|<--- c --->|
identifier
Otherwise, code 000 in field b to indicate a new instruction fetch is required, and code xxx0...0 at the beginning of the next physical storage word.
leading field of next instruction

Figure 7: **"Fitting"** Syllables at the End of a Storage Word

Intuitively, this gains some of the spatial advantage of Huffman like codes (at word boundaries) for the simple straight binary code, yet permits rapid decode. In theory, it could also be used in conjunction with more highly encoded forms (either SDL or pure Huffman): the relative time gain would be smaller since decode overhead would dominate the instruction fetch, however; and the space gain wowld be reduced due to the reservation of the all zeros code. Time and space estimates for this form are:

Access Time = (k+l) * (R*w/b + shift(w) + test)

Program Space = w * (k+l)

R, k, b, and w are again the same as for the threaded code and record oriented code cases; "shift(x)" is the number of host instructions required to extract an x bit field; and "test" is the number of host instructions needed to check for the all zero code (which should be zero in a well designed DEL host).

3.2.2.5. Field Dependencies

So far, we have discussed only static dependencies. It is also possible to take advantage of locality by dynamically changing the interpretation of specific codes. That is, the semantics associated with special DEL operators may be used to change the tables used by the decode routine within the interpreter. While this generally requires rather sophistocated compilation techniques (see Foster and Gonter [9], and Sweet [28]]), it may be possible to avoid exhorbitant overhead by applying this stratagem only when DEL control passes from one module to another.

This is because of the one-to-one correspondence between DEL modules and the lexical "scopes" in the source programs from which they were derived. Fixing the size of an operand reference upon entry to a DEL module can result in dramatic compression of program size, and should be considered when synthesizing a DEL for any block structured source language. Applying this technique to operator references is more difficult, since there is no direct semantic correlation between the set of operators applied in a module and the definition its scope.

Conceivably, escape codes could be used to reduce the number of bits required to distinguish between individual DEL operators. As far as the interpreter is concerned, the only cost of such conditional operator codes would be the inclusion of distinct operator decode tables for each escape class. Explicit escape codes may have to be inserted at every potential target of an unstructured GOTO, however, which will increase both the time and space required during execution.

A similar problem is encountered when generating register oriented DEL surrogates, where the values of individual variables must be saved before executing an unstructured GOTO, and restored upon arrival at each potential target of a GOTO. Discussion of the flow analysis techniques required to improve on this naive strategy is beyond the scope of this work (see Geshke [10], Elson and Rake [5], McKeeman [22] and [23]). Our concern is with the underlying structure and form of a DEL.

3.3. The Action Rule

As mentioned in the first section, the action rule consists of a function applied over a domain of arguments that produces one result. There are two considerations in synthesizing an action rule: format and operation.

The synthesis objectives for both considerations should be clear from the earlier discussion of cannonic interpretive form:

* Enough formats should be available to provide transformational completeness;

* Each HLL operation should have a corresponding interpretation within the limits of interpreter size.

3.3.1. Formats

In order to recognize and interpret DEL instructions, the interpreter must be able to determine the size and meaning of at least the next syllable to be fetched and decoded. The leading syllable in an instruction usually specifies its layout and interpretation; i.e., defines the format of the instruction.

In order to select an optimal format set in an orderly manner, it is necessary to first construct a universe of formats that at least covers the combinatorial bindings found in traditional zero, one, two, and three address architectures. For the moment, we need only distinguish between two general classes of operand references: explicit reference, which appear as distinct syllables within an instruction; and implicit references, which are defined by the instruction's format code.

We use a three letter mnemonic code to describe associations of implicit and explicit operands with at most two arguments and one result (binary order). The first letter identifies the operand to be bound to the left argument of the operator (if any); the second letter identifies the operand to be bound to the right argument (if any); while the third letter identifies the operand to be bound to the result (if any). Seven letter designations are sufficient to describe all relevant possibilities:

1. "S", an implicit specification of the cell just above the top of the evaluation stack (value denoted by <u>s</u>).
2. "T", an implicit specification of the cell that was the top of the evaluation stack (value denoted by <u>t</u>).
3. "U", an implicit specification of the cell just below the top of the evaluation stack (value denoted by <u>u</u>).
4. "A", the first explicit operand specification appearing in an instruction (value denoted by <u>a</u>).
5. "B", the second explicit operand specification appearing in an instruction (value denoted by <u>b</u>).
6. "C", the third explicit operand specification appearing in an instruction (value denoted by <u>c</u>).
7. "_", for <u>null</u>, meaning "not applicable" -- probably due to low functional order.

A use ordered analogue to the typical 360/370 instruction "AR R1 R2" (meaning "add registers R1 and R2, leaving the result in R1") would be written "ABA R1 R2 +" in this notation. A zero address DEL expansion for the same operation might appear as: "AS R1 :=; AS R2 :=; UTU +; TA R1 :=".

This notation also covers various hybrid formats that use both implicit and explicit references in a single instruction; for example, the use ordered hybrid instruction "TAB X Y -" means "subtract the value of X from the value currently on top of the dynamic evaluation stack, store the result in Y, and decrement the stack pointer" (top of stack is always defined with reference to its state before interpreting the formate in question).

It is easy to identify the characteristic formats for traditional zero (UTU), one (TAT), two (ABA), and three (ABC) address architectures using this system. The restrictive nature of these mono format DELs is clear in comparison to the 343 potential formats designations suggested by our three letter mnemonic.

The obvious implementation for all of the formats suggested by this identification scheme, however, would require 7*7*7 distinct interface routines and 9 bits per instruction (assuming a straight forward, n-way binary encoding). Even if the spatial cost were acceptable in the DEL program space the associated interface routines would occupy too great a fraction of micro store for most host machines. Consider the following rules for eliminating formats that are redundant with respect to our notion of transformational completeness cited in the canonic interpretive form discussion.

1. Formats violating standard LIFO stack accessing conventions are not required (this would eliminate such formats as UAB, STU, ABU, etc.).
2. Only one ordering of T and U in the first two (argument) positions is needed--we use the UT ordering, which is consistent with a left to right, depth first post order taversal of the macro-tree representation of a program.
3. Formats that differ only by a permutation of explicit references are equivalent (e.g., ABC, ACB, BCA, BAC, CBA, and CAB are all equivalent; we choose the alphabetized element, ABC in this case).
4. Formats differing only by a permutation of the null designator, "_", in the first two (argument) positions are equivalent--we use formats with a leading null.

All of the above elimination rules can be applied without adversely affecting either the compilation or execution phase. Using these rules, the 343 element format universe suggested by our combinatoric identification rule can be reduced to 30 elements. The table below lists all distinct combinations remaining after these rules have been applied, grouped in order of increasing functional order.

The branches in a macro definition tree [5] may be thought of either as explicit references (if connected to a leaf node), or as implicit references (if connected to an ancestor node). This establishes a connection between format structure and the context of operator nodes in a macro definition tree. By inspection, at least one of the above formats is directly associated with each possible configuration of an ancestor node.

Table of Potential Formats

MNEMONIC	TEMPLATE	SEMANTICS	STACK
___	<OP>	call op	
S	<OP>	s := op	+1
A_	<X> <OP>	x := op	
_T_	<OP>	call op(t)	-1
_A_	<X> <OP>	call op(x)	
TT	<OP>	t := op(t)	
-AS	<X> <OP>	s := op(x)	+1
-TA	<X> <OP>	x := op(t)	-1
AA	<X> <OP>	x :? op(x)	
AB	<X> <Y> <OP>	y := op(x)	
UT	<OP>	call op(u,t)	-2
TT_	<OP>	call op(t,t)	-1
AT_	<X> <OP>	call op(x,t)	-1
TA_	<X> <OP>	call op(t,x)	-1
AA_	<X> <OP>	call op(x,x)	
AB_	<x> <Y> <OP>	call op(x,y)	
UTU	<OP>	u := op(u,t)	-1
TTT	<OP>	t := op(t,t)	
UTA	<X> <OP>	x := op(u,t)	-2
TTA	<X> <OP>	x := op(t,t)	-1
TAA	<X> <OP>	x := op(t,x)	-1
ATA	<X> <OP>	x := op(x,t)	-1
TAT	<X> <OP>	t := op(t,x)	
AAS	<X> <OP>	s := op(x,x)	+1
TAB	<X> <Y> <OP>	y := op(t,x)	-1
ATB	<X> <Y> <OP>	y := op(x,t)	
AAB	<X> <Y> <OP>	y := op(x,x)	
ABB	<X> <Y> <OP>	y := op(x,y)	
ABS	<X> <Y> <OP>	s := op(x,y)	+1
ABC	<X> <Y> <Z> <OP>	z := op(x,y)	

While we have reduced the spatial requirements of multi-format DEL structures to a practical order of magnitude, implementing all 30 formats listed in the table may still be prohibitive for some hosts. The following theorems identify some interesting subsets of this format universe.

Theorem 1: The canonic interpretive form requirements can be satisfied using only eleven formats, up to the level of diadic operators, if "reverse" forms for all non-commutative operators are included in the set of action functions.

Proof: Consider the following DEL restrictions and interpreter coding conventions.

1. Semantic routines for monadic operators mwst increment the pointer to the top of the DEL evaluation stack before performing their normal processing.

2. "Reverse" forms for all non-commutative (diadic) operators must be included in the repertoire of DEL action functions.

Given these restrictions, we may eliminate all format codes whose mnemonic contains the "_" by using the binary format containing a "S", "T", or "U" in the same position, but which is otherwise identical (interpreter convention). Formats differing only by a reversal of the left and right argument binding (e.g., ABA and ABB) are redundant under the DEL restriction; only one element of each such pair is needed. Finally, no format whose code begins with "TT" can be generated by a naive compiler, since this would require recognition of the use of an intermediate value as a repeated argument.

The set {UTU, UTA, TAT, TAA, TAB, AAS, ABS, MA, AAB, ABA, ABC} satisfies the theorem by inspection.

Theorem 1 demonstrates that the individual advantages of both stack and register oriented architectures can be merged at a gross cost of only four bits per instuction, which compares favorably with typical polish DELs (in which each instruction contains two form bits to distinguish between "push", "pop", "operate", and "literal"). For example, a single TAB format is equivalent to the polish sequence "push A, operate, pop B"; the first requires one instruction and four format bits, the second requires three instructions and six format bits.

Determining the relative advantage of a format rich DEL over a mono format, register oriented DEL with a variety of addressing modes is more complicated. Auto increment and decrement capability can be used to simulate a stack architecture, while indexing and indirecting can be used to simulate memory to memory oriented architectures. Addressing mode flexibility does not extend to exploiting multiply used operands, however, and is manifestly not as compact or efficient as an implicit stack architecture (it is difficult to perform net adjustments to the stack pointer, for example). Further, as will be seen in the next section, there are more direct operand reference encodings that can be used on most dynamic hosts.

Theorem 2: Only four formats are required if the DEL evaluation stack is eliminated.

Proof: The set {AAA, AAB, ABA, ABC} is sufficient, by inspection.

Compilation is somewhat more difficult in this case, however, since "dummy" variables must be synthesized in order to evaluate compound expressions. Although fewer bits would be needed to indicate the format code, it is likely that the space and time required during execution would increase because of these extra explicit operand syllables.

Theorem 3: Only six formats are needed to satisfy all but the "unique variable" requirement of the canonic interpretive form.

Proof: The set (UTU, UTA, TAT, TAB, ABS, ABC} is sufficient, again by inspection.

It is difficult to determine whether or not execution phase time and space would increase or decrease if this reduced format set is used, however, since the question is sensitive to user behavior. The smaller format sets are interesting because of their coding compatibility with hosts strongly biased toward 8 bit storage quanta. If only two or three bits are needed to define the format of an instruction, then it is possible to combine both the format and operator code in a single byte.

Any of the above format sets would be enhanced by the addition of special formats to handle reverse forms of non commutative operators (e.g., ATT, ATA, ATB, and ABB), or of auxilary formats to simplify interface processing for unary operators (e.g., TT, TA, AS, AA, and AB). One or two "escape" formats might also be added to provide a mechanism for implementing higher order formats (for operators with greater than binary order), user defined operators, or other DEL extensions. The critical point is that these format sets are "rich" enough to guarantee that no non functional, memory oriented overhead instructions need be generate or evaluate arithmetic expressions: i.e., their M-ratio is zero by construction.

3.3.2. Selecting Operators

Suppose that the design of a DEL is complete except for the selection of its operator set; and further that a finite set $F = \{f_i\}_{i=1}^{N}$ of potential operators is "well known" -- in the sense that there s a <u>micro expansion</u> x_i and a <u>macro expansion</u> X_i for each potential operator f_i $(i=1, \ldots, N_F)$. Intuitively, x_i is the body of a host routine that implements the semantics of f_i, while X_i is constructed entirely from operators in the set $\{f_j\}_{j \neq i}$ -- and so could be generated in place of f_i should it <u>not</u> be selected as a DEL operator. The problem is to find a subset G of F that minimizes the space and time requirements of the resulting DEL.

Let w_i be the number of micro store words required by x_i, and W be the total number of words of micro store that can be used to hold semantic routines. The difficulty is that $w_1 + w_2 + \ldots + w_n$ may be greater than the number of available words of micro store, so that it is not possible to simply set G equal to F. Let:

d_i = the dynamic frequency of f_i;
t_i = the average time needed to execute x_i;
T_i = the average time needed to execute X_i;
s_i = the static frequency of f_i;
l_i = the length of the identifier for f_i;
L_i = the length of X_i;

and for any subset Z of F, define:

$$t(Z) = d_1*t_1 + . . . + d_n*t_n;$$
$$T(Z) = d_1*T_1 + . . . + d_n*T_n;$$
$$s(Z) = s_1*l_1 + . . . + s_n*L_n;$$
$$S(Z) = s_1*L_1 + . . . + s_n*L_n;$$
$w(Z)$ = the sum of all w_k such that f_k is an element of G; and
$$E(Z) = -(\ (A*(t(G) + T(F-G)) + B*(s(G) + S(G))\ ).$$

The intent is to quantify the notion of efficiency by a linear function, E -- which implies that the marginal utility of micro store is constant. This is a reasonable approximation for small changes in the DEL operator set, since only a small fraction of the total space available would be affected. The objective is now to find a set G that maximizes E, subject to the constraint $w(G) < W$. To this end, define the merit of selecting operator f_i (i.e., the incremental advantage of placing semantic routine x_i in micro store) to be:

$$m_i = A*(d_i*(T_i-t_i)) + B*(s_i*(L_i-l_i))$$

Further, let the merit m(Z) of any subset Z of F be the sum of the individual merits m_i for all i such that f_i is an element of Z. It can be assumed without loss of generality that the elements of F are ordered such that i < j implies either $m_i/w_i > m_j/w_j$, or $m_i/w_i = m_j/w_j$ and $w_i < w_j$. The claim is that this defines a natural lifeboat ordering for F; as reflected by:

Theorem 4: If G is the subset $\{f_1, f_2, \ldots f_n\}$ of F such that $w(G) < W < w(G) + w_{n+1}$, then
$$W(H) < W \Rightarrow E(H) - E(G) < m_n*(W-w(G))$$
for any subset H of F.

Proof: Let H be any subset of F satisfying the hypothesis. If GH denotes the intersection of G and H, then $w(H-GH) < W - w(G) + w(G-GH)$ by definition[10]. Now, $m_j/w_j \le m_n/w_n$ for all j such that f_j is in H-GH since j must be greater than n by construction; this means that:

1) $$m(H-GH) < (m_n/w_n)*w(H-GH) < (m_n/w_n)*(W-w(G)+w(G-GH))$$

since $w_j > 0$ for all j. But $(m_n/w_n)*w(G-GH) < m(G-GH)$, again by construction; this means

2) $$m(H-GH) - m(G-GH) < (m_n/w_n)*(W-w(G))$$

Since m(Z) = E(Z) + A*T(F) + B*S(F) for any subset Z of F,

3) $$E(H) - E(G) < (m_n/w_n)*(W-w(G))$$ qed.

The difference in efficiency between an optimal DEL and that resulting from an application of Theorem 4 must be less than a comparatively small factor (m_n/w_n) times the unused micro store (W-w(G)). The product should be quite small in comparison to the overall efficiency rating of the DEL -- both because W-w(G) is small in comparison to w(G), and because m_n/w_n may be no greater than m_i/w_i for all $i < n$.

The practical simplification is that it is no longer necessary to formulate and solve a general linear programming problem in order to select an efficient operator set. The question of how F is determined, however, remains open. In many cases it is probably sufficient to set F equal to the set of all functions used in the semantic specification of the given source language. If the highest performance is to be achieved, however; additional operators are likely to

----

[10] $w(X-XY) = w(X) - w(XY)$ for any subsets X and Y of F.

be needed. The following principles may be useful; let F_0 be a preliminary set of operators derived by inspection of the source language semantics:

1) Set F_0 equal to the set of primitive functions extracted by inspection of the semantic specification for the given source language.

2) Form F_1, the closure of F_0 under n-ary composition (n = l-3 should be sufficient in light of Knuth's statistics [17]).

3) Form F_2 by including natural decompositions for complex functions (e.g., extracting "normalize" and "unnormalized multiply" operators from a standard "floating multiply").

4) Form F_3 by including special operators for frequent bindings of operators in F_2 to literal arguments (e.g., adding a unary "INC" operator to replace "_+1"), and again taking closure.

In general, it is important to exploit implicit specification of functions or arguments whenever possible -- a typical example being the automatic invocation of a "standard fix-up" after arithmetic overflow or underflow. This is especially true of program control and data conversion/selection operators. For example, if the source language is strongly structured[11], then it may be possible to keep a stack of pertinent variables, addresses, etc., within micro store to speed up the execution of looping constructs and/or recursive procedure invocation.

----

[11] I.e., all control structures are strictly one-in one-out.

As a case in point, consider a generalized END0 operator that controls termination of FORTRAN DO-loops. This operator requires four operands: an iteration count variable (J); an increment value (I); a maximum count (M); and a loop transfer label (L). The expansion for a typical loop, "DO 10 J=N,M,I", might be:

```
        MOVE <N> <J>
L       (body of loop}
        ENDO <J> <I> <M> <L>
```

In this implementation, the iteration count variable is explicitly initialized prior to loop entry. The ENDO operator must bind the identifiers <J>, <I>, and <M> to the appropriate values; increment J by I; and compare the result to M, performing the appropriate data-dependent branch for each iteration. There is no way to avoid the initialization data-dependent branch steps, but if there are no explicit transfers in or out of the loop body, special initialization and termination operators could be used:

```
        INITDO <N> <J> <I> <M>
L      (body of loop)
        ENDX
```

In this case, the INITDO operator would temporarily move the values of J, I, and M into micro store, initializing J in the process. The loop-back address would also be automatically initialized at this point. The ENDX operator need not repeatedly fetch, decode, and bind the identifiers for J, I, M, and L to their respective values. This saves four field extractions and four variable accesses per iteration (the value of J must be both loaded and stored).

3.4. Process Name Space--General Issues

A name used by a process is a surrogate for a value. The set of all names that can be accessed by a process is the name space for that process. Source level names are usually just alphanumeric strings imbedded within a program text; DEL level names are operand identifiers appearing within executable instructions (usually in l-l correspondence with source names); and host level names are simply addresses of accessable elements of the host storage' hierarchy. Values are associated with names via a "contents map"--at any point during a computation, the contents of a name is its correct value. In this discussion, we are concerned only with the properties of names themselves, not with the form of identifiers for these names or the problem of interpreting identifiers within an executable instruction; the contents mapping is assumed to be established externally--e.g., by a loader.

Some issues related to the concept of a process name space are:

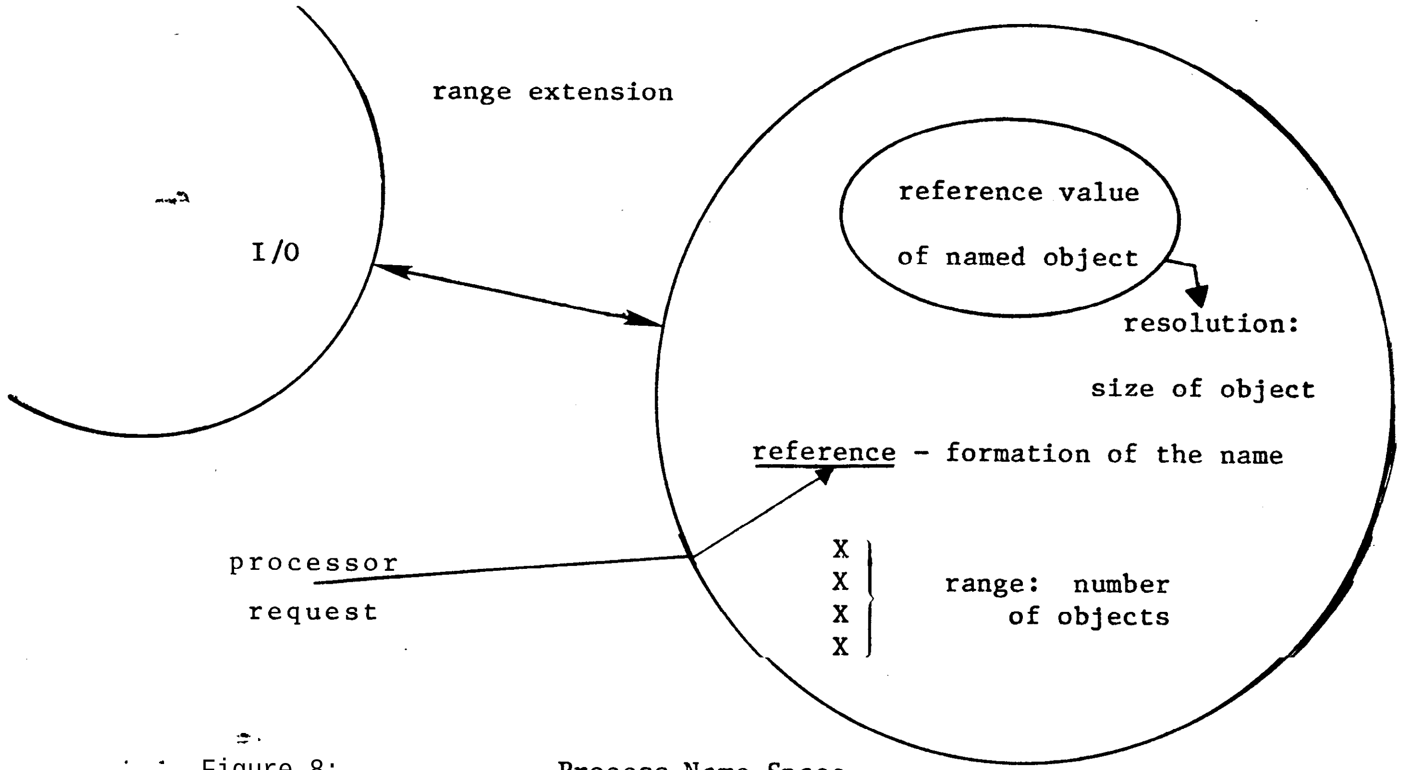


Figure 8: Process Name Space

1. range and resolution of objects,
2. range extension--I/O handling and files,
3. homogeneity of the space,
4. reference coding.

Range and Resolution:

Range and resolution refer to the maximum number of objects that can be specified in a process space and the minimum size of an object in that name space respectively. Traditionally, instructions provide resolution usually no smaller than an 8 bit byte, and frequently a 16 bit or larger word, and range defined as large as one can comfortably accomodate within the bounds of a reasonable instruction size and hence program size. Thus, ranges from 2^{16} for minicomputers to 2^{24} for System 360 include most common arrangements.

Range Extension:

The range of the name space directly accessable to a host is bounded, so it is essential that an extension mechanism be provided to allow a process to access large data bases (e.g., I/O and file handling). If the directly accessable range were unlimited, then as soon as objects were entered anywhere in the system, the place of entry in the processor name space could be regarded as an element in the process name space.

An associated problem is that of attaching records to an established process name space. Usually this attachment must be done by a physical movement of data from its present location to an area within the bounds of the present process name space before it can be operated on. The programmer must manage data movement from the I/O space into the process name space through I/O commands. This binding or attachment is the responsibility of the programmer and must be performed at the correct sequential interval so as to insure the integrity of the data and yet not exceed the range limitations of the name space--overflow buffers, for example. Ability to communicate between an unbounded I/O media and a bounded processor name space allows the programmer to simulate an open ended name space.

It is, however, an uncomfortable requirement placed on the programmer, and frequently results in cumbersome and inefficient operations. Of course, the larger the range, the more precise and variable the resolution, the easier it is to manage objects in the process name space; flexibility in this regard both permits and promotes conciseness during program development.

OBSERVATION: From the above, the desirability of an unbounded name space with flexible attachment possibilities is clear.

Homogeneity:

While name spaces may be partitioned in many different ways, homogeneity refers to partitions distinguished by the action rule of a process. Action rules or instructions generally cannot treat all objects in the same way. Certain classes of objects are established such as registers, accumulators, and memory objects. Action rules are applied in a non-symetric way: one of the arguments for an action rule must be a register whereas the other may be a register or a memory object. The premise of this partitioning is performance, i.e. the assumption that access to registers is faster than access to memory. Thus, many familiar machines have their name space partitioned into a register space and memory space: 360, PDP-11, etc. As the partitioning of the name space increases, its homogeneity decreases.

References:

Mapping identifiers into their image in the host name space--i.e., determining the actual location or address of a named object--involves a subtle series of design issues. There is a broad spectrum of potential tradeoffs between interpretation time and program representation size. Traditional issues in identifier construction include: short vs. long addresses, indexing; indirection; dynamic tagging; etc.

The reference problem may be broken down into two parts, referencing operands and referencing operators. Operand referencing involves extracting or updating the value of an object, while operator referencing involves the invocation of an action rule (i.e., process state transformation).

3.4.1. Name Space Synthesis

Providing a flexible and effective name space structure helps minimize the space and time requirements of a DEL. Good designs are characterized by both a simple correspondence between the source name space and the DEL name space (to simplify compilation and preserve transparency), and a simple correspondence between the DEL name space and the host name space (to maintain efficiency during execution).

High level language name spaces generally involve effectively unbounded ranges, one dimensional reference structures (viewing subscripted arrays and other qualified references as "expressions" rather than primitive symbols), and discrete granularity (i.e., reference structure does not induce a fixed relation between referands in the memory space). The identifiers used as references at this level are syntatically homogeneous, but semantically inhomogeneous--i.e., interpretation of the contents map for a referand depends on the context in which its reference appears. In particular the referand associated with a particular source name may be different for different occurrences of that name.

This is because the name space of most source programs is partitioned into distinct scopes of definition (or "scope" for short; intuitively, a scope is simply a natural grouping of references within which the association between references and referands is fixed, unless altered explicitly by dynamic allocation or redefinition statements).

On the other hand, most host level name spaces are structurally inhomogeneous, being partitioned into register sets, storage modules, etc. References to elements in these partitions are rarely interchangeable within a host instruction. The association between references and referands is usually fixed at this level, however, even though it may be parameterized in terms of the current contents map (e.g., as in indexed or indirect referencing). Such discrepancies between the source and host name spaces account for much of the difficulty in synthesizing an effective DEL name space.

DEL organizations may be classified according to the placement of different portions of the information needed to bind a reference to a referand (Chevance [3]). Data is characterized by three distinct pieces of information: type, locator, and value. The type of a referand defines the range of values it may assume; its locator defines the address to be used when accessing its contents; and its value is the bit pattern assigned by the current contents map, which must be interpreted according to its data type.

The type and locator may be specified either in the operation code of an instruction or in operand reference codes, either directly or indirectly (e.g., through a display vector). Four such combinations are:

1. Type in operation code, locator in one dimensional reference (conventional machine languages).
2. Type and locator concatenated in two dimensional reference (this form is typical of higher level DELs--e.g., Weber [29], Wilner [31], and Wortman [32]).
3. Type and locator concatenated in a "descriptor" identified indirectly through a one dimensional reference (descriptor based machines).
4. Locator is reference indirected individually through a two dimensional reference (theoretical, no known example).

The traditional approach is to partition the DEL name space along the same lines as the host name space, mapping symbolic names into distinct indexed (two or three dimensional) references; i.e., a type 1, 2, or 4 organization. The compiler must insure that the proper base address is loaded into an appropriate index register when the translated references are evaluated. This increases the M-ratio of the resulting dynamic instruction stream by requiring significant load/store activity to maintain correct base register values. For example, the statement "I = J - I" might expand to:

```
L    R1, @I
L    R2, @J
L    R2, 0(R2)
S    R2, 0(R1)
ST   R2, 0(R1)
```

using a 360/370 machine language DEL. Only the subtract instruction is functional; the first and second instructions are overhead caused by the range differential between source and DEL name space, while the third and fifth instructions are memory oriented overhead caused by a combination of the inhomogeneity of the DEL name space (storage and register references no interchangeable) and combinatorial restrictions of the 360 architecture (it has no ABB format). This approach emphasizes the importance of register allocation, and leads to elaborate multi pass algorithms for minimizing load/store activity (Sethi [25] and Stockausen [27]).

Incorporating locator information in the reference itself also leads to complications in handling the thorny problems associated with changes in scope (e.g., storage management, passing parameters, and accessing externally defined referands); none of the above forms solves this problem by construction. Perhaps the best known model for describing the effects of scope is the Contour Model developed in Johnson [16]. This model is rich enough to describe the address map transformations required by the allocation, release, and retention rules of most sowrce languages, and captures all practical methods of binding actual arguments to formal parameters as well. Its guarantee of completeness suggests the Contour Model as a good design base for DEL name spaces.

A process is defined to be a time invariant algorithm together with a time varying record of execution. Discrete points in an execution record are identified by an encoded pair, formal parameters in a different manner than local variables, however, either by including explicit operators in the DEL instuction stream (McClure), or always testing for indirection (Wilner)--Bashkow avoids the problem by restricting his source language to a subset that does not include subroutine blocks or arrays.

3.4.2. Environment and Contours

The notion of environment is fundamental not only to DELs but also to traditional machine languages as evidenced by widespread adoption of cache and virtual memory concepts. What is proposed here is akin in some respects to the cache concept and yet quite distinct from it. We recognize locality as an important property of a program name space and handle it explicitly under interpreter control. Thus, locality is transparent to the DEL name space but recognized and managed by the interpreter. Properties of the environment are:

1. The DEL name space is homogeneous and uniform with an a priori unbounded range and variable resolution.

2. Operations, involving for example the composition of addresses which use registers, should not be present in the DEL code but should be part of the interpreter code only. Thus, the register name space and the interpreter name space are largely not part of the DEL name space and it is the function of the interpreter to optimize register allocation.

3. The environment locality will be defined by the higher level language for which this representation is created. In FORTRAN, for example, it would correspond to function or subroutine scope.

4. Unique to every environment is a scope which includes:

   i. a label contour,
   ii. an operand contour,
   iii. an operation table.

Following the Johnston model, we define a contour to be a vector (or table) of object descriptors. When an environment is invoked, a contour of label and variable addresses must be established (if not already present) in the interpretive storage. For a simple static language like FORTRAN this creation can be done at load time. For languages that allow recursion, etc., the creation of the contour wowld be done before entering a new environment. An entry in the contour consists of the (main memory) address of the variable to be used; this is the full and complete DEL name space address. Type information and other descriptive details may also be included as part of the entry.

The environment must provide a pointer into the current contour, and must define the width of identifiers for labels and variables. Typically, the contour pointer and identifier width would be maintained in the register of the host machine. We denote identifier width by W and the pointer to the base of the current contour by EP; Figure 9 illustrates the process of referencing a DEL entity using this terminology. Both labels and variables may be indexed off the same environmental pointer. Subfields within DEL instructions, then, are actually containers for immediate values that define indices in the current contour; contour entries at the indexed location define the mapped address of the desired variable or label in the host name space. In other words, the operand identifiers within DEL instructions are simply contour indices that select a particular description for the image of a given source level object in the host name space.

The Contour Model differs from other high level DEL architectures in that the function of references is separated from that of descriptors. References are one dimensional indices into a current

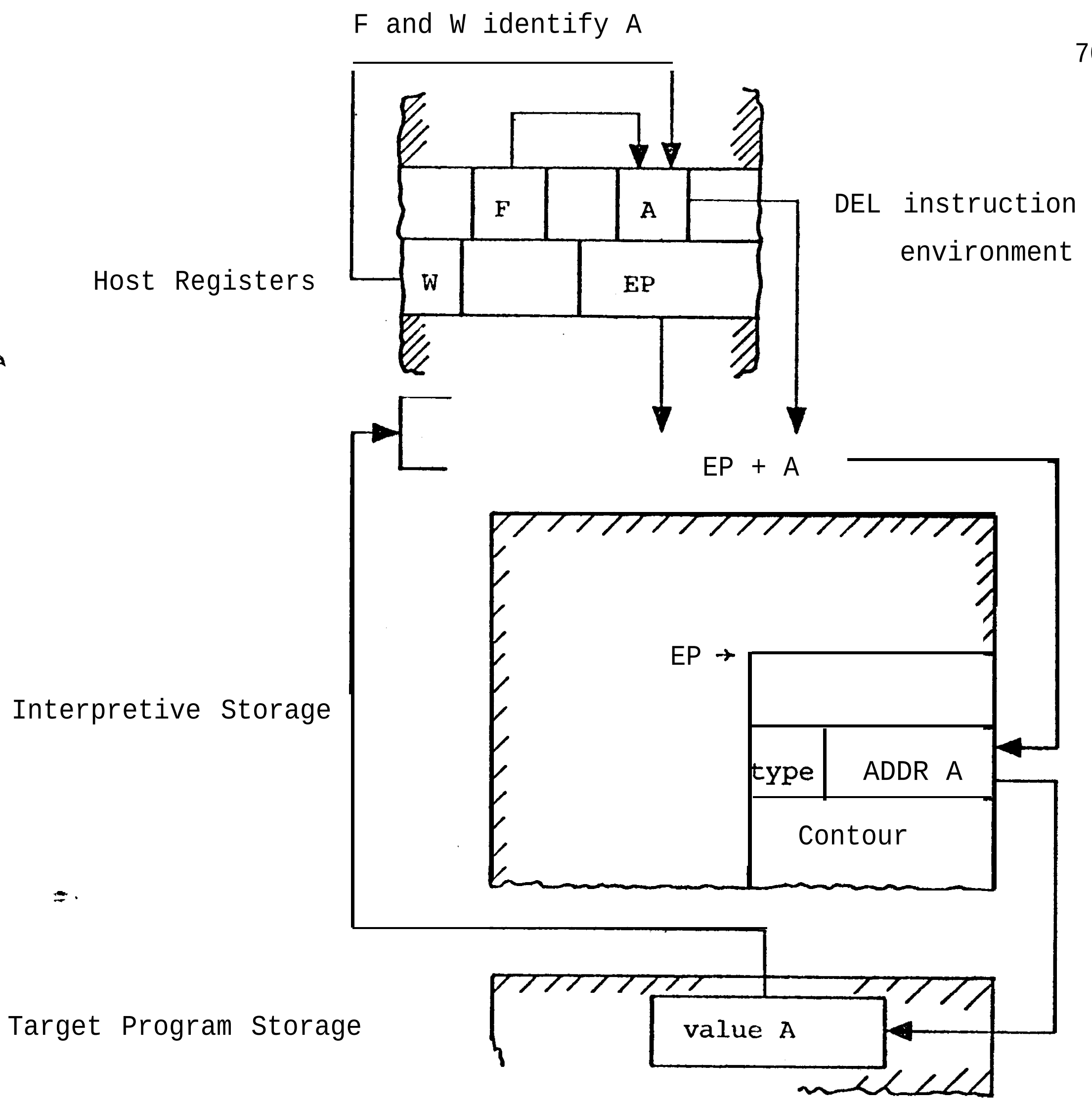


Figure 9: Referencing a DEL Variable

declaration array, which we call the current contour. The current contour is always maintained within the host micro store, and a new contour is created for each distinct incarnation of a source scope. This is an extreme case of a type 2 organization, in which only W bits are used to represent a reference--where W is the smallest integer such that there are less than 2^W distinct referands in the current access environment.

Each contour is uniquely identified by an environment pointer that, at least logically, denotes its zeroth element. The environment pointer for the current contour is part of the DEL program state vector, and must be saved/restored when entering/leaving a scope of definition. The address map is computed by adding the reference code to the current environment pointer, and then accessing the appropriate **referand** descriptor (Figure 10):

descriptor (reference N) = micro store (ep + N)

value (reference N) = main store (descriptor N)

Figure 10: Normal DEL Addressing Structure

This analysis can be extended by noting that the logical type of a **referand** (integer, floating point, logical, or character) can be separated from its physical type (single, double or varying **perci**sion). We refer to the physical type as "shape". Elements of contours are descriptors, each of which is itself a vector that defines the **shape**, **type**, and locator of a **particlar** DEL entity--or, more precisely, the algorithm used to access that entity. Distinguishing shape within the descriptor allows us to use semantic routines designed for the general case, rather than having one per type:shape combination.

It is important that descriptor processing be kept as simple as possible. For most languages, this means that the value of the variable will be located in the main store cell whose address is defined by--the appropriate descriptor--e.g., the value of the n-th DEL variable is located in the memory cell(s), whose initial address is given by the contents of the n-th word of the contour in micro store. If this is done, then the effective address of a referand can be calculated in two basic host cycles using our method (micro store is assumed to have an access time comparable with the time needed to perform a primitive arithmetic operation). Essentially, dynamic contours are a simple mechanism for exploiting the writability of modern micro stores; in effect we have created a distinct 'base register' for each 'distinct DEL entity rather than for contiguous blocks of entities.

If the source language has the property that two distinct source names can never denote the same referand, then the indirection step may be avoided by maintaining values of (scalar) DEL variables directly in the contour. This is not usually the case, however; due either to "overlay" capability (e.g., the EQUIVALENCE feature in FORTRAN, or pointer references in PASCAL), or to the possibility of binding the same actual argument to two distinct formal parameters using "call by reference" or "call by name".

Given a fully static source language (like BASIC, FORTRAN, or PASCAL) a unique contour for each distinct scope of definition may be preallocated during compilation. In this case, only the descriptors for formal parameters need be modified during execution. For most source languages, however, a new contour will have to be created each time a new scope is entered; particularly if the source language supports recursive procedure invocation. In this case, a highly encoded header could be attached to the algorithmic body of DEL surrogates to serve as a phantom, or "skeletal" contour. Descriptor components that can be fixed at compilation would appear as literals in this header; components that can not be determined until block entry would be parametrically encoded to simplify run time computation.

Since the header entries need be evaluated only once per contour creation, they can be relatively complex and difficult to evaluate. However, this factors out the common calculations needed to compute effective addresses; there will be a substantial time savings whenever variables are accessed repeatedly within a contour, and the possibility of a time loss when variables are not accessed at all. The penalty can be avoided by marking descriptors in the current contour as "unbound" until they are actually referenced. Each time a DEL reference is processed, its descriptor must be checked for validity; this usually means that some form of hardware support is required for this stratagem to work efficiently. Lacking a tagged architecture, it is likely that the time needed to decide whether a contour element is a value or a descriptor will swamp the time saved by sometimes avoiding a main store access. The "tag" in this case is not a type field concatenated with values in main store, but rather a "presence flag" concatenated with the descriptor/value in micro store. This keeps the number of tag bits low, and simplifies host implementation. Such an explicit caching technique should be evaluated carefully in light of the specific capabilities of the given host.

The contour technique is easily adapted to most existing parameter passing conventions. Parameters may be passed "by reference" simply by copying the appropriate descriptors from the caller's contour into the callee's contour. Parameters are passed "by value" by initializing a variable created either in the caller's environment (call by copy value), or in the callee's environment (call by value copy), with the value of the argument referand in the caller's contour. "By name" parameter passing involves moving an IP:EP pair into the appropriate descriptor in the callee contour; the IP:EP, where IP is an instruction pointer into the time invariant algorithm, and EP is an environment pointer identifying a particular access environment. No transformation identified by the IP can depend upon or alter the contents of a memory cell unless that cell is in the address mapping image of the current access environment.

Every access environment contains a declaration array that is, conceptually, a linear vector of address map definitions. Each entry in the declaration array is uniquely associated with a particular source name, and completely specifies all of the information needed to access the referand of that name. In practice, the Contour Model is usually realized in terms of a two dimensional reference structure of the form level:offset, where "level" is associated with a lexical scope of definition and "offset" is associated with the physical location of a referand (level codes are also called segment numbers, block numbers, page numbers, etc.; and offset codes are sometimes referred to as occurrence numbers or placement indices).

Upon entering a scope, a block of storage is allocated in the memory space sufficient to contain all of the local variables known to be referenced within the block. During compilation, various positions relative to the beginning of this block are preassigned to specific source referands--thus determining the offset code for their associated references. Storage is usually managed by partitioning it into two distinct classes: a LIFO stack that contains all of the local referands allocated automatically at scope entry; and a heap that contains all referands that exist independently of the normal procedure entry/exit mechanism.

The obvious space saving aspect of linear contours is that only W bits are needed to identify an arbitrary DEL variable. Only three or four bits are needed to encode W within the DEL program status vector so that it could easily be updated each time the environment pointer is changed, allowing the inherent locality of well structured source programs to be exploited in a direct manner. This method is at least as fast as the display vector approach--and may well be more efficient since it does not incurr multiple decode overhead, since it involves only a one dimensional index.

3.4.3. Operation Contours

Each verb or operation in the higher level language identifies a corresponding interpretive operator in the DEL program representation (control actions may be treated either as an operation or as a format type). The routines for interpreting all familiar operations are expected to lie in interpretive storage. Certain unusual operations, such as transcendental functions, may not always be contained in the interpret storage. A pointer to an operator translation table must be part of the environment; the actual operations used are indicated by a small index container off this'pointer (Figure 11). The table is also present in the interpretive storage. For simple languages, this latter step is probably unnecessary since the total number of operations may be easily contained in, for example, a six bit field and the saving in DEL program representation may not justify the added interpretive step.

In general, contours could be established for DEL blocks corresponding to: a single source operator; an individual source statement; a linear segment of source statements; a source clause; a source procedure; or the entire source program. Further research is required to determine which level is space-time optimal. It should be

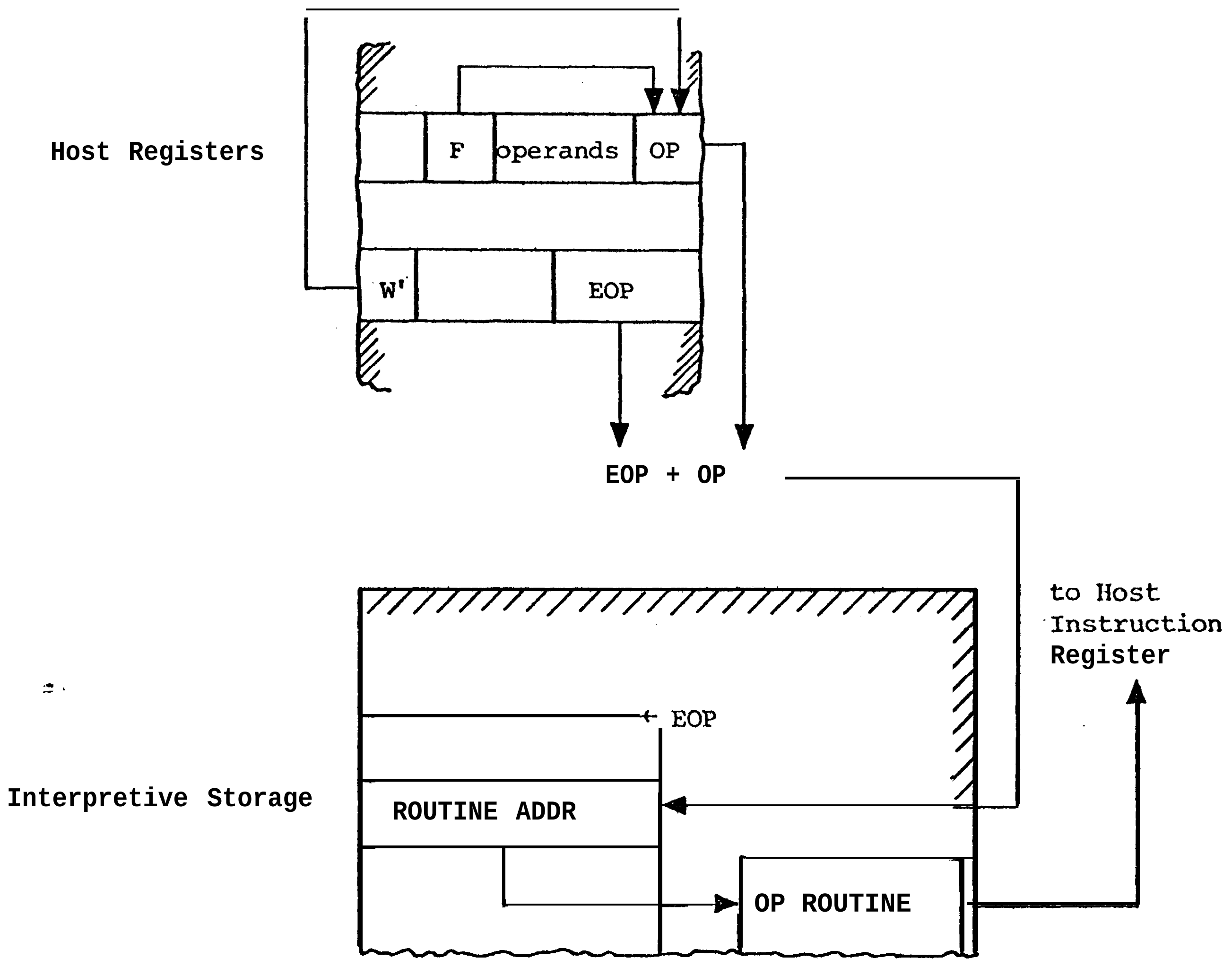


Figure 11: Referencing a DEL Operator

noted, however, that loop and procedure blocks are reasonable choices for contour extents: a significant amount of non-trivial sequential processing must be performed to enter or exit these constructs, which affords at least the opportunity to overlap contour creation with other mandatory computations.

3.4.4. AN EXAMPLE AND SOME RESULTS [8]

Again consider the previous example:

```
1  I  =  I+1
2  J  =  (J-1)*I
3     K= (J-1)*(K-I)
```

This might be implemented as:

Statement	Implementation				Semantics
	4	2	2	2	
1	ABA	I	1	+	I := I+1
2	ABT	J	1	-	T := J-1
	TAB	I	J	*	J := T*I
3	ABT	J	1	-	T := J-1
	ABT	K	I	-	T := K-I
	TUA	K	*		K := T*U

where T and U are the top and next-to-top (under top) stack elements, respectively. The size, in bits, of each identifier field in the first instruction appears directly above the corresponding mnemonic. Note that the stack is "pushed" automatically by the 5th instruction and the 6th instruction "pops" the stack for further use.

Our CIF rules apply directly to container size--two bits are allowed to identify the four variables and two bits are used for the four operations. The canonic number of instructions are achieved, as are the variable and operation container sizes; however, 4 additional bits per instruction are needed in this implementation to identify the correct format (out of the eleven instruction formats discussed in Theorem 1, plus four additional control operators).

There is a difference between the transformational completeness required by the canonic rules, and the achieved transformational completeness. The two agree only for statements containing at most one functional operator--so that the implementation contains an additional J-identifier in instruciton 3 and an additional K-identifier in instruction 6. These do not, however, necessitate additional memory references since separate domain and range references are also required in the CIF if a single variable is used both as a source and sink within a given statement. The comparison with the CIF measures are shown below.

ACHIEVED vs. THEORETICAL EFFICIENCY

Number of	Achieved	CIF
Instructions	6	6
Operand Identifiers	11	9
Operator Identifiers	6	6
Memory References	12 2 (i.u.) (data)	12 1 (i.u.) (data)
Totals	---- 14 total	---- 13 total

Size of	Achieved	CIF
Each Identifier	2 bits	2 bits
Total Program	58 bits	30 bits

We assume that 32 bits are fetched per memory reference during the instruction fetch portion of the interpretation process. While the program size has grown with respect to CIF measure, it is still substantially less than System 370 representation; other measures are comparable to CIF.

The example discussed in the preceding section may be criticized as being non-typical in its DEL comparisons:

i. The containers are quite small, thus reducing size size measures for the DEL code.

ii. Program control is not included.

iii. The program reduction in space may come at the expense of host machine interpretation time.

With respect to the first criticism, note that the size of a program representation grows as a log function of the number of variables and operations used in an environment. If sixteen variables were used, for example, program size would increase by 50% (to 90 bits). It is even more interesting, however, to observe what happens to the same three statements when they are interspersed in a larger context with perhaps 16 variables and 20 statements and compiled into System 370 code. The size of the object code produced by the compiler for either optmized or unoptimized versions increases by almost exactly the same 50%--primarily because the compiler is unable to optimize variable and register usage.

The absence of program control also has no significant statistical affect. A typical FORTRAN DO or IF is compiled into between 3 and 9 System 370 instructions (assuming a simple IF predicate) depending upon the size of the context in which the statement occurs. Thus, the inclusion of program control will not significantly alter the statistics and may even make the DEL argument more favorable.

The third criticism is more difficult to respond to. We submit that host interpretation time should not be noticeably increased over a traditional machine instruction if the same premises are made, since

i. 16 DEL formats must be contrasted against perhaps 6 or 8 System 370 formats (using the same definition of format)--not a significant implementation difference.

ii. Some features are required by a 370 instruction even if not required by the instruction--e.g., indexing. Name completion through base registers is a similar situation since the base values remain the same over several instructions.

iii. Approximately the same number of state transitions are required for either a DEL instruction or a traditional machine instruction if each is referred to its own "well mapped" host interpreter. In fact, for an unbiased host designed for interpretation the interpretation time is approximately the same for either a DEL instruction or a System 370 instruction.

The language DELtran, upon which the aforementioned example was based, has been developed as a FORTRAN DEL. The performance and vital statistics of DELtran on the host EMMY [24] is interesting, especially when compared to the 370 performance on the same system. The table below is constructed using a version of the well-known Whetstone benchmark and widely accepted and used for FORTRAN machine evaluation. The EMMY host system referred to in the table is a very small system--the processor consists of one board with 305 circuit modules and 4096 32 bit words of interpretive stcrage. It is clear that the DELtran performance is significantly superior to the 370 in every measure.

DELtran vs. System 370 Comparison for the Whetstone Benchmark

Whetstone Source	-- 80 statements (static)
	-- 15,233 statements (dynamic)
	-- 8,624 bits (excluding comments)

	System 370 FORTRAN-IV opt 2	DELtran	ratio 370/DELtran
Program Size (static)	12,944 bits	2,428 bits	5.3:1
Instructions Executed	101,016 i.u.	21,843 i.u.	4.6:1
Instructions/Statement	6.6	1.4	4.6:1
Memory References	220,561 ref.	46,939 ref.	4.7:1
EMMY Execution Time (370 emulation approximates 360 Model 50)	0.70 sec.	0.14 sec.	5:1
Interpreter Size (excludes I/O)	2,100 words	800 words	2.6:1

Before concluding, a further comparison is in order, Wilner [31] compares the S-language for FORTRAN on the B-1700 as offering a 2:1 space improvement over System 360 code. The FORTRAN S-language instruction consists of a 3 or 9 bit OP code container followed by operand containers of (usually) 24 bits--split as descriptor, segment and displacement (not unlike our interpretive storage entry). The format set used in this work is of limited size, and does not possess transformational completeness. However, even this early effort offers noticable improvement of static program representation.

References

[1] Bashkow, Theodore R., "System Design of a FORTRAN Machine," IEEE Transactions on Electronic Computers, Vol. EC-16, No. 4, August 1967, pp. 485-99.

[2] Burroughs Corportation, B1700 Systems Reference Manual, Burroughs Corporation, Detroit, Michigan, 1972.

[3] Chevance, R. J., "Design of High Level Language Oriented Processors," SIGPLAN Notices, Vol. 12, No. 1, January 1977, pp. 40-51.

[4] Control Data Coporation, Control Data 600 Series of Microprogrammable Prcessors (Reference Manual), Control Data Corporation, Minneapolis, Minnesota, 1972.

[5] Elson, M., and Rake, S. T., 'Code-generation techniques for large-language compilers," IBM Systems Journal, Vol. 9, No. 3, 1970, pp. 166-88.

Et- Flynn, Michael J., 'Trends and Problems in Computer Organizations," IFIPS Congress, Stockholm, Sweden, August 1974, North Holland Publishing Company, 195, pp. 2-10.

[7] - , and Henderson, D. S., "Variable Fied-Length Data Manipulation in a Fixed Word-Length Memory," IEEE Transactions on Electronic Computers, Vol. EC-12, No. 5, October 1963, pp. 512-17.

[8] , "The Interpretive Interface: Resources and Program Representation in Computer Organization,' Proceedings of the Symposium on High Speed Computers and Algorithm Organization, University-of Illinois, Champaign Illinois, (Pub. Academic Press) April 1977.

[9] Foster, C. C., and Gonter, R. H., "Conditional interpretation of operation codes," IEEE Transactions on Computers, Vol. C-20, No. 1, January 1971, pp. 108-11.

[10] Geshke, Charles M., Global Program Optimization, Ph.D. Thesis, Carnegie Melon University, Pittsburg, Pennsylvania, October 1972.

[11] Green J., "Microprogramming, emulators, and programming languages," Communications of the ACM, Vol. 9, No. 3, March 1966, pp. 230-31.

[12] Hoevel, Lee W., "DELtran Principles of Operation," Digital Systems Labratory, Technical Note No. 108, Stanford University, Stanford, California, March 1977.

[13] - , with Wallach, Walter A., "A Tale of Three Emulators," Technical Report No. 98, Digital Systems Laboratory, Stanford University, Stanford, California, October 1975.

[14] Huffman, D. A., "A method for the construction of minimum redundancy codes," IRE, Vol. 40, No. 9, September 1952, pp. 1098-101.

[15] International Computers Limited, CRIL Reference Manual (internal draft), Kidsgrove, England, 1971.

[16] Johnston, John B., "The Contour Model of Block Structured Processes," Procedings of the SDSPL (SIGPLAN Notices, Vol. 6), February 1971, pp. 55-82.

[17] Knuth, D. E., "An Empirical Study of FORTRAN Programs," Software--Practice and Experience, Vol. 1, 1971, pp. 105-33.

[18] - , The Art of Computer Programming, Vol. I (Fundamental Algorithms), Addison-Wesley, Reading, Massachusetts, 1968.

[19] Lawson, Harold W., Jr., "Programming-Language Oriented Instruction Streams," IEEE Transactions on Computers, Vol. C-17, No. 5, May 1968, pp. 467-85.

[20] Lunde, A., "Empirical Evaluation of Some Features of Instruction Set Processor Architectures," Communications of the ACM, Vol. 20, No. 3, March 1977, pp. 143-52.

[21] McCarthy, J., et al., LISP 1.5 Programmer's Manual, MIT Press, Cambridge, Massachusetts, 1965.

[22] McKeeman, W. M., "Language Directed Computer Design," Proceedings of the Fall Joint Computer Conference, Vol. 31, Fall 1967, pp. 413-17.

[23] - , et al., A Compiler Generator, Prentice Hall, Englewood Cliffs, New Jersey, 1970.

[24] Neuhauser, Charles J., "An Emulation Oriented, Dynamic Microprogrammable Processor (Version 3)," Technical Note No. 65, Digital Systems Laboratory, Stanford University, Stanford, California, October 1975.

[25] Sethi, Ravi, and Ullman, Jeffrey D., "The Generation of Optimal Code for Arithmetic Expressions," Journal of the ACM, Vol. 17, No. 4, October 1970, pp. 715-28.

[26] Standish, T. A., "A Preliminary Sketch of a Polymorphic Programming Language," Centro de Calculo Electronico, Universidad Nacional Autonoma de Mexico, July 1968.

[27] Stockhausen, Peter F., "Adapting Optimal Code Generation for Arithmetic Expressions to the Instruction Sets Available on Present-Day Computers," Communications of the ACM, Vol. 16, No. 6, June 1973, pp. 353-54 (short communication).

[28] Sweet, Richard Eric, Empirical Estimates of Program Entropy, Ph.D. Dissertation, Department of Computer Science, Stanford University, Stanford, California, December 1976.

[29] Weber, Helmut, "A Microprogrammed Implementation of EULER on IBM System/360 Model 30," Communications of the ACM, Vol. 10, No. 9, September 1967, pp 549-58.

[30] Wichman, B. A., "Five Algol Compilers," Computer Journal, Vol. 15, No. 1, January 1972.

[31] Wilner, W. T., "Burroughs B1700 memory utilization," Proceedings of the Fall Joint Computer Conference, Fall 1972, pp. 579-86. --

[32] Wortman, Daniel B., A Study of Language Directed Computer Design, Ph.D. Thesis, Stanford University, Stanford, California, 1973.